AFRL-RI-RS-TR-2009-271
Final Technical Report
December 2009

TRUST-MANAGEMENT, INTRUSION-TOLERANCE, ACCOUNTABILITY, AND RECONSTITUTION ARCHITECTURE (TIARA)

Massachusetts Institute of Technology

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2009-271 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
LOK K. YAN	EDWARD J. JONES, Deputy Chief
Work Unit Manager	Advanced Computing Division
	Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* December 2009	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* April 2007 – June 2009

4. TITLE AND SUBTITLE TRUST-MANAGEMENT, INTRUSION-TOLERANCE, ACCOUNTABILITY, AND RECONSTITUTION ARCHITECTURE (TIARA)	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-07-2-0032
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Howard Shrobe, Andre DeHon, Thomas Knight	**5d. PROJECT NUMBER** NICE
	5e. TASK NUMBER 00
	5f. WORK UNIT NUMBER 06
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology, 77 Massachusetts Ave, Cambridge MA 02139-4301; University of Pennsylvania, 200 S. 33rd Street, Philadelphia PA 19104	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITA, 525 Brooks Rd., Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2009-271

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-5045-3 Dec 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the Trust-management, Intrusion-tolerance, Accountability, and Reconstitution Architecture (TIARA) system, a broad design effort including novel computer architecture, operating system and application middleware. TIARA illustrates that a highly secure computer system can be designed without sacrificing performance. TIARA involves three major sub-efforts: A hardware security tagged architecture (STA) that tags each word of the computer's memory with metadata such as the data type and compartment of the data. The STA hardware enforces access rules controlling which principals are allowed to perform which operations on which data. This allows the construction of a novel Zero-kernel Operating System (ZKOS) that has no single all privileged kernel and that provides strong guarantees against penetration. Finally TIARA provides a level of application middleware that enforces architectural level constraints and maintains the provenance of application data. All common exploits are preventable by the TIARA architecture and this incurs only a minor increase in chip area.

15. SUBJECT TERMS
Tainting, tagged, metadata, architecture, hardware, processor, microkernel, zero-kernel, co-design

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Lok K. Yan
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	133	**19b. TELEPHONE NUMBER (*Include area code*)** N/A

Contents

List of Tables

List of Figures

1 Executive Summary

Today's computer systems are practically and fundamentally insecure. Our adversaries (criminals, terrorists, rival states) repeatedly demonstrate that the security we deploy is not adequate to protect the valuable information our systems contain. The roots of this problem are not a mystery: (1) Our hardware is based on outdated design compromises and does not provide the right semantic foundation, (2) the core of our software systems deliberately compromise sound security principles for performance and time-to-market concerns, and (3) the computer system artifacts we build are beyond our science and engineering to analyze and manage. Exponential growth of hardware capacity, decrease in costs, and deployment of computer-mediated interactions have outstripped the development of the sound engineering necessary to assure that we have control of our computer systems and can manage the risks they entail.

While the situation today is dangerous, it does not have to remain that way. Some of the same forces that exacerbated today's problem (*e.g.* exponential growth in hardware capacity) enable alternate solutions. By re-evaluating computer system design in light of today's inexpensive silicon hardware, today's state-of-the-art in software design and assurance, and the criticality of today's information processing systems, we can engineer systems that we can analyze and trust.

In particular, if we are not beholden to legacy processor and operating system designs, we can build a new hardware base that embraces good security principles (complete mediation, least privilege, separation of privileges), does not compromise performance for security, does not depend on large-scale, error-free software, and makes automated verification of critical security properties tractable.

This report describes Trust-management, Intrusion-tolerance, Accountability, and Reconstitution Architecture (TIARA) a codesign of new hardware, operating system (OS), and application infrastructure. The hardware base, the Security Tagged Architecture (STA), carries rich metadata throughout runtime to assure the semantics of the computation are preserved and supports the security principles at the level of the hardware. This eliminates the traditional pressure to build monolithic, over privileged security domains and allows the design of a Zero Kernel Operating System (ZKOS) embodying a radical decomposition of the operating system into many small components each with limited privileges. It decreases the opportunity for compromise of each OS component, makes it possible to formally verify each component, and assures containment so that even if a component is compromised, it does not allow the system to be subverted.

A clean-slate redesign of the hardware and software foundation for computer systems is a large task. In this short, limited scope effort our goal was to increase the confidence that such a task is worth undertaking, reducing the risks and uncertainties that come with pursuing a new and alternate path. As such, we concentrated on the areas of greatest concern (*e.g.* will this hardware be too expensive or too slow?, can software be decomposed to eliminate single points of failure?).

In this effort, we have demonstrated:

- Complete mediation and metadata propagation at the primitive processor word-level requires hardware investments that are tiny compared to today's processors and need not impact cycle times (Section 5.6).
- Separation of privileges can be applied at the level of primitive services. It appears increasingly plausible that the operating system can be decomposed to avoid giving any entity complete authority. We demonstrate this in key areas of memory management (Section 4.4), device drivers (Section 4.5), scheduling (Section 4.3), and logging (Section 4.2.1).
- A set of patterns can be used to structure services in ways that support this secure decompo-

sition, simplify programming, and enable verification. We describe such a key design pattern allowing strong decomposition and tight controls on information flow (Section 4.1).

- The hardware enables application-level access control, operating at the level of the method call, that need not be a burden to the programmer or compromise performance (Section 7.1). In addition to access control, these facilities include, information flow tracking, provenance maintenance and execution monitoring of data flow, control flow, and enforcement of the input-output invariants of application methods.

We have further developed or refined a number of key ideas:

- Tags and rule enforcement serve as hardware interlocks that can be used as a safety net around software (Section 2.3). This has the effect of significantly reducing the size of the description that must be assessed to validate security properties. This is a key part of making formal, mechanical verification possible.
- Metadata-driven hardware interlocks make it practical to take the security principles of Saltzer and Schroeder [59] seriously (Section 2). Cycle-by-cycle enforcement of metadata rules allows us to apply them at the most primitive level of object representation. This support in turn makes it possible to apply them at the level of operating system services and applications, realizing separation patterns like Clark-Wilson [16].

Based on our experience and developments, we recommend:

- It is time to stop being complacent about the current hardware security base. Fully developing a new architecture for deployment is a large effort that will take time and resources. It is time to get started.
- It is time to give up on the hypothesis that we can write large, error-free software systems. Instead, we should develop and employ engineered systems that can achieve the necessary trustworthiness despite the inevitable residual flaws in systems. The new hardware base suggested above makes this possible. This means moving away from relying on monolithic domain software systems as our security base. We must develop a new base system around ideas like those in ZKOS (Section 4) and build the infrastructure (tools, libraries, languages) to allow applications to be structured in this way as well.
- It is time to stop relying on systems that exceed both human and machine understanding. No single human can manage millions of lines of code, and no formal system can validate unstructured code of this size and complexity. Formal verification is now powerful enough to be used, but we must design and decompose our systems to meet the capabilities of verification.
- The hardware, OS, and verification strategy must be designed together. Engineered codesign is within the realm of state-of-the-art engineering, while analysis and assurance of unconstrained hardware and software artifacts is not.

2 Overview of Technical Approach

TIARA involves a clean-slate redesign of the processor architecture along with a novel operating system that has no single all privileged principal (*i.e.*, no kernel). This eliminates a host of security vulnerabilities, changing the tradeoffs associated with security and performance, and transforming cyber conflict by shifting the advantage to the defenders. Our novel Security Tagged Architecture for the TIARA processor

1. Enforces the semantics of the computation,
2. Allows fine-grained assignment of privileged operations,
3. Provides lightweight isolation of fine-grained memory regions.

Exploiting these unique capabilities, TIARA's Zero Kernel Operating System decomposes software components (including those normally federated into an operating system kernel) into small, isolated, cooperating components that have minimal and distinct privileges and are mutually suspicious of each other. The resulting system thus embodies key principles identified by Saltzer and Shroeder [59]:

- *Complete mediation*: Every access to every object must be checked for authority—the STA hardware mediates *both* (a) the semantic validity of every operation and (b) the authority on every instruction performed and every word of data.
- *Least privilege*: Each module is granted only the minimal privileges necessary to do its job—this can be controlled to the level of individual privileged processor instructions on specific data types and words of memory.
- *Separation of privilege*: Protection mechanisms should require that more than one condition should be met before access is permitted. More generally modules should be distrustful of one another and check one another as in [16]. This provides breach and error containment—rather than a single breach giving complete access to unrelated systems, this makes it necessary to compromise (or find errors in) a collection of components in order to subvert a system.

By spending a modest amount of, now cheap, additional hardware, we provide this dramatic increase in security without sacrificing performance.

Before going deeper into the details of our technical approach, we first examine the root causes of the broad classes of insecurities in today's computer systems.

2.1 Software Insecurity Arises from The Lack of Enforced Semantics

All modern commercial operating systems are vulnerable. Figure 1 shows the rate at which attacks have been growing [15] while [39] documents that, despite years of patching, the skill level required to launch an attack has been **decreasing** due to accumulated tool development and software engineering by the attacking community.

In a previous project, we collected a catalog of vulnerabilities in the Firefox browser[1]. As shown in Figure 2, nearly all of the vulnerabilities arise from a failure of the underlying hardware and software to enforce the semantics of object extent (*e.g.*, buffer overflows), identity (*e.g.*, storage

[1]Conducted as part of the DARPA Application Communities program

management bugs that lead to "dangling pointers"), and type (*e.g.*, faulty method dispatch caused by passing an integer to a routine expecting an object).[2]

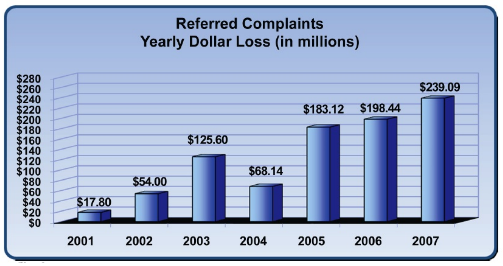


Figure 1: The Number of Successful Attacks is Growing Rapidly

Category	Description	N
Stack Overflow	Ill formed data causes overwriting of stack frame with data that is then branched to for execution	2
Array Access	Reference to data outside bounds of array which is then branched to for execution	2
Heap Overflow	Ill formed data causes overwriting of heap with data that is then branched to for execution	11
Dead Pointer	Use of an invalid pointer to inject data that is then branched to for execution	3
Trampoline Errors	Passing of invalid data to method dispatch routine causes branch to arbitrary position in memory	2
Garbage Collection	Violation of memory conventions causes garbage collector to create dead pointers	13

Figure 2: Vulnerabilities in Firefox

Current computing hardware and software systems are extremely vulnerable precisely because they violate the principles cited above. Processors provide a single privilege mode (kernel) that has complete authority over the system, memory systems provide isolation only by separating large address spaces, it is expensive (thousands of cycles) to switch between address spaces, software (including the operating system) is organized into large address spaces of code and data where any weakness can be exploited to access or subvert the whole, and the hardware and system software treat the memory as consisting of "raw seething bits" whose meaning is not understood or respected.

No element of the entire stack is responsible for enforcing over-arching conventions of memory structuring or access control; nothing enforces the procedure call (and general stack use) abstraction/encapsulation. Outsiders may easily penetrate the system by exploiting vulnerabilities (e.g. buffer overflows) arising from this lack of basic constraints. Attacks are not easily contained, whether they originate from the clever outsider who penetrates the defenses or from the insider who exploits existing privileges. Because there are no facilities for tracing the provenance of data, even when an attack is detected, it is difficult if not impossible to tell which data are traceable to the attack and what data may still be trusted. Because there is no tracing of accountability, insider attackers have limited reason to fear discovery. Similarly, the lack of data provenance makes it difficult to tell what information must be removed in order to downgrade the classification of a document.

Because there is so little inherent structure, the core of the operating system must be protected from other parts of the OS and especially from application layer code. This is typically done by creating a barrier between the kernel and the rest of the software, in which the kernel operates in a separate address space as does each user process. However, the cost of the context switch between kernel space and user space is normally quite expensive, as is the cost of interactions between separate user processes. As a result the system winds up as a large monolithic kernel possessing many disparate facilities all of them sharing unlimited privileges. In addition, this results in a complex computational model with many mechanisms for interactions between user processes and the OS.

In summary, in today's systems:

[2]The systematic use of a type safe language (*e.g.*, Java, Lisp, ML) could remove many of these vulnerabilities. Ironically, Firefox is implemented in a type safe language (Javascript), but much of the runtime library for Javascript is written in C from which it inherits vulnerabilities.

- Violation of intended semantics allows attackers to subvert the system (*e.g.* write beyond the end of an object, overwrite the words on stack that should not be visible or addressable, treat a data word as an indirection address or branch target, branch to and execute words that are not instructions, perform a method dispatch on data that is not a class instance).
- Once one error is exploited, the attackers can take control of the entire system (*e.g.* access to read or modify all memory, access to all privileged operations (*e.g.* rewrite page tables, change permissions, access devices, install software)).

This state of affairs arises, in part, from system architectures and engineering trade-offs and a mind-set grounded in the realities of an earlier period in which computer resources were relatively scarce and it was expensive to provide enforcement mechanisms for the underlying semantics of the computation. Hardware and system software treat memory as "raw seething bits," provide few or no mechanisms for checking types or bounds, and place the burden of storage management on the programmer. This lack of the most basic semantic enforcement is then compounded by an equally serious lack of tools for managing notions of locality, separation, privilege and access rights that are necessary to limit the effects of a penetration.

Even as the VLSI (Very Large Scale Integration) revolution provided increasing resources, the need to fit a whole processor (together with its caches, multiple pipelines, speculative execution units, etc.) on a single die still dictated that chip real estate was expensive and that the most important design trade-offs were those that increased performance. In the context of scarce resources, attempts to optimize the performance at all costs eclipsed the quest for a simple elegant architecture that delivers safety and trustworthiness. With today's abundant resources, we should now return to the task of optimizing for these qualities.

Ultimately, modern processors provide only a single mechanism for privilege and separation management: the user/kernel distinction managed by virtual memory hardware that provides page level read/write/execute access controls on a per process basis. While this is valuable, it operates at the wrong granularity, controlling access to individual pages of virtual memory rather than to the semantically more meaningful unit of the individual object. Because the kernel is all privileged, a penetrated kernel component can be used as a vehicle to compromise any other resource in the system. Even processors and OS'es that provide for more levels of protection, such as multiple rings [51, 65] still structure these in a strict hierarchy of increasing privilege; penetration to ring 0 leads to unlimited privilege. There have been recent attempts to address this problem (*e.g.*, SELinux (Security Enhanced Linux) [29] and MILS (Multiple Independent Layers/Levels of Security) [1]) by providing OS mechanisms for enforcing mandatory access controls, but these mechanisms are limited by the lack of fine-grained, non-hierarchical hardware support. Similarly, operating systems provide "access control lists" (ACLs) and the like to control file permissions; these also operate at a coarse grain and can be subverted by gaining "root access."

Due to the large amount of state information that must be exchanged to safely hand off the processor from one principal to another (*e.g.*, special processor state, register file, Translation Lookaside Buffer (TLB) content), context switches are relatively expensive (thousands of cycles) motivating system programmers to move code into the kernel to limit the number of domain crossings. In practice, the kernel becomes bloated and violates the principle of least privilege [59]. Each component within the kernel enjoys unlimited privileges even though it typically needs only a very limited subset of these privileges.[3]

[3]Much research has been conducted on alternatives, but in practice the overhead of context switching between

The lack of structure in the hardware and operating systems forces the system programmer to trade-off security against performance. Performance concerns have generally trumped security concerns.

The last 20 years have led to unprecedented improvements in chip density and system performance, fueled mainly by Moore's Law. During the same time, system and application software have bloated, leading to unmanageable complexity, vulnerability to attack, rigidity and lack of robustness and accountability. Moore's law has mainly been used to allow us to move applications to increasingly smaller and cheaper platforms (e.g. mini-computers, workstations, personal computers and personal data assistants). At the time that these platforms are introduced they are limited in resources; thus, out our engineering designs have continued to be dominated by a sense of scarcity in which there is no reasonable alternative to having all key elements of the computational environment, from hardware through system software and middleware to application code regard the world as consisting of unconstrained "raw seething bits".

The design of today's systems violated reasonable principles as a tradeoff to achieve acceptable performance in an era of scarce hardware resources. However, decades of Moore's Law hardware scaling have delivered at least three orders of magnitude greater hardware capacity today than when the architecture of these systems was selected. With a competent processor now small compared to a silicon die (*c.f.* quad-core processors from Intel and AMD), we now live an era of abundant hardware resources. These conventional systems are based on *now obsolete* technology assumptions and "laws" for hardware/software design. Today's multicore chips and massive field programmable gate-arrays make clear that processor architects are no longer starved for real-estate. Rather they are starved for good ideas about how to use the abundantly available on-chip resources. These dramatic advances in hardware capabilities and the equally important progress in formal methods over the past 3–4 decades present a qualitatively different tradeoff space.

This has led us to adopt TIARA's **key design principle** that it makes sense to commit a modest portion of these now abundant hardware resources to eliminating security vulnerabilities by enforcing basic computational semantics and making isolation and privilege separation sufficiently inexpensive that it can be used freely to provide strong separation (*e.g.* [1, 50]). A strong, secure and robust computing base can be built without sacrificing performance by using some of these newly available resources to build in far greater support for the semantics of the computation; in particular, we argue that we can easily afford to carry metadata and access control rules into the runtime of the system and that this information can make it clear what the computation is allowed to do. Novel hardware techniques can make the performance cost of processing this metadata nearly zero.

2.2 Host Security Is Essential to Network Security

So far, we have been emphasizing the weaknesses of host hardware, operating systems and application software. Nevertheless, most discussions on the national stage have centered on the term "network security". This is actually a somewhat vague and often misleading term, implying that the problem lies in the network infrastructure (e.g. routers, DNS servers, TCP/IP protocols). However, a careful consideration of what has been termed "network security" shows that a better term would be security of networked computers. The vast majority of vulnerabilities that have been exploited are not in the network infrastructure but rather in host software. Consider the Conficker worm, one of the most serious recent network attacks, described in the quotation below [41] from the New

kernel mode and normal mode is so high that it places a high premium on moving functionality out of the kernel.

York Times.

> In recent weeks a worm, a malicious software program, has swept through corporate, educational and public computer networks around the world. Known as Conficker or Downadup, it is spread by a recently discovered Microsoft Windows vulnerability, by guessing network passwords and by hand-carried consumer gadgets like USB keys.
>
> Experts say it is the worst infection since the Slammer worm exploded through the Internet in January 2003, and it may have infected as many as nine million personal computers around the world.
>
> Worms like Conficker not only ricochet around the Internet at lightning speed, they harness infected computers into unified systems called botnets, which can then accept programming instructions from their clandestine masters. “If you’re looking for a digital Pearl Harbor, we now have the Japanese ships steaming toward us on the horizon,” said Rick Wesson, chief executive of Support Intelligence, a computer security consulting firm based in San Francisco.

At the time of this article, Conficker had more than five million computers under its control including government, business and home computers in more than 200 countries, according to the New York Times article.

As the article notes, the vulnerability exploited by Conficker was found in the Microsoft Windows operating system, not the network infrastructure. The same is true for the “Slammer” worm mentioned above and for many other large scale attacks. The SANS list of top vulnerabilities [61] includes the following:

- Client-side Vulnerabilities in:
  - Web Browsers
  - Office Software
  - Email Clients
  - Media Players
- Server-side Vulnerabilities in:
  - Web Applications
  - Windows Services
  - Unix and Mac OS Services
  - Backup Software
  - Anti-virus Software
  - Management Servers
  - Database Software
- Security Policy and Personnel:
  - Excessive User Rights and Unauthorized Devices
  - Phishing/Spear Phishing
  - Unencrypted Laptops and Removable Media
- Application Abuse:
  - Instant Messaging
  - Peer-to-Peer Programs
- Network Devices:
  - VoIP Servers and Phones
- Zero Day Attacks:

  - Zero Day Attacks

From this it is clear that the vast bulk of the problems are in host software, followed by operations practices; only a very small fraction of vulnerabilities lie in network devices, services and protocols.

The significance of the network rather lies in the fact that:

- The network provides an attacker with remote (and often anonymous) access to vulnerable hosts.
- The network amplifies host vulnerabilities:
  - Insecure hosts are subverted to work for the attacker and serve as a platform for attacking other hosts.
  - Even within a fire-walled intranet, by subverting a single machine the attacker establishes a beachhead for controlling the entire intranet.
  - By allowing ensembles of compromised hosts to act in concert to recruit and subvert other machines anywhere on the Internet, the network becomes the vehicle for enabling National-scale Zero-Day attacks.

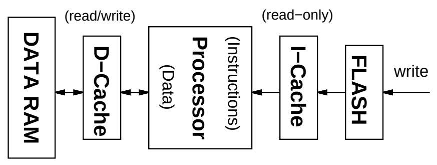
The existence of exploitable host vulnerabilities in machines connected by the Internet allows attacks of previously unimagined consequences. The key enabler is the vulnerabilities in host software, not the vulnerabilities in the network infrastructure, which mearly acts as an amplifier.

Thus, the core technical challenge is to fix the vulnerabilities in host software and to make it much more difficult to use whatever vulnerabilities remain as the foothold in national zero-day attacks. The goal of the TIARA project is precisely to attack this head-on by providing hardware tools that allow fundamentally new ways of structuring host software. The new structures are much more likely to be constructed without vulnerabilities, to be analyzable and verifiable, and to require multiple independent penetrations by an attacker in order to gain control.

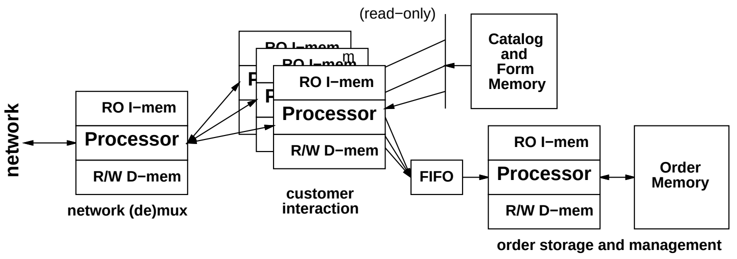
2.3 Mitigating Risks with Hardware Interlocks

Consider the problems that might arise in trying to build a web-based e-commerce system in which purchases require customer payment and shipping data. Today's common practice is to build systems from reusable software components, where the components may be legacy software, outsourced, freeware, or purchased libraries. Since such servers are extremely performance-sensitive, many components might be combined in a single address space, losing whatever protections are inherent in the virtual address protection mechanisms of the processor. Hence, an error (*e.g.*, the sadly still-common buffer overflow) in any component creates opportunities for an attacker to gather customer credit card numbers, either directly or via code injected to 'bootstrap' more sophisticated exploits. Injected code can also serve as a beachhead from which to mount attacks on the server operating system, which, like the web server, is a large and complex software system composed from many pieces from various sources.

Securing a code base of this scale would require capabilities far beyond the state of the art in either software engineering or formal methods. The bottom line problem is that **the hardware's protection mechanisms and the software engineering needs are mismatched**, and the complexity of the aggregate of the two inhibits use of even the most aggressive of today's formal methodologies. Is the solution to be found in the software, verification, or hardware domain?



(a) Harvard Architecture



(b) Internet Merchant Front End

Figure 3: Hardware Isolation for Internet Merchant

As a thought experiment (following [56]), consider the stronger protections that a hardware architect might propose. As an example, with code and data separated as shown in Figure 3(a) and certain assumptions (*e.g.*, that code is compiled to machine code and no run-time code generation), we can use specialized memory technologies to prohibit inappropriate data transfers. For example, to prevent writing code, we can position an I-cache (Instruction cache) in front of a flash memory writable only via physical access, or we can segregate memory and processing for orders taken by our web merchant with a one-way FIFO (First In First Out) path that prevents data from propagating to the Internet (Figure 3(b)).

In this hardware isolation model, we have "spent" hardware mechanisms to achieve a considerable gain in security. We can easily reason about information flow at a coarse-grained level in such a system, and, without analyzing any software, we can begin to make guarantees about what flows are and are not possible.

However, this solution has some unattractive features, in addition to the substantial additional hardware. First, it is very problem specific (*i.e.*, it may not be useful for any application beyond the web merchant's). Second, it has relied on limiting the capabilities of the software. What is instead desired is a *general purpose*, configurable coupling between software and hardware that permits fine-grained analysis and hardware protection and that enforces the desired information flow architecture at a very fine-grained (*e.g.*, memory reference) level.[4]

A tagged processor can provide isolation without compromising flexibility. A processor that carries metadata encoding types and compartments with every word and that uses a Tagged Memory Unit (TMU) to enforce access rules on each cycle can be programmed to enforce the same type of strong flow restrictions as the special-purpose hardware solution just described. Each compartment emulates a separate memory, while the access rules model the constraints on mutability of the data in that memory and the limitations on communications channels with other memories. The tagged memory processor is both cheaper and more flexible than an application-specific design with multiple special-purpose memories; it can be configured to support any application needs. The software architecture and its intended isolation can be enforced *directly* with hardware, operating in a single address space and consequently avoiding context switches.

This assures that information flow questions are decidable and reduces the size of the description we must scrutinize and rely upon for correctness. That is, security now depends on the rules we enforce on the flow of information in the system. The rule sets are not Turing Complete making all the questions we ask about them decidable, whereas questions

[4]Rushby's original paper [56] led in the direction of separation kernels [1]. STA can be see as a hardware-based separation kernel.

about software are undecidable in the general case. This ruleset is also very small compared to the size of the code in the system. In a conventional, software-only system, security depends on the correctness of *all* the software. Here, we factor out the information-flow specification into a much smaller description that is tractably verifiable independent of the entire body of software. In fact, the software can have bugs, and the rule specification prevents those bugs from compromising the overall security properties of the system.

To ground the reduction in size, a typical system will need 1–10 rules to specify how data flows over a link between two connected processes, and a process may exchange data with 5–10 other processes. This means, at most 100 rules (and more likely 10) govern a single process. This should be contrasted with the tens of thousands of lines of code that may be required to describe the computational task for the process.

2.4 Overview of The TIARA Architecture

TIARA starts with the premise that we can use some of today's abundant computational resources to fix critical security problems using a combination of hardware, system software, middleware and programming language technology. TIARA is less vulnerable, more tolerant of intrusions, capable of recovery from attacks, and accountable for its actions. TIARA's design imposes minimal impact on overall system performance.

The TIARA architecture achieves these goals through the judicious use of a modest amount of extra, but general purpose, hardware (the *Tag Management Unit or TMU* shown in figure 4) that is dedicated to tracking the security context of data at a very fine grained level, to enforcing access control policies, and to constructing a coherent object-oriented model of memory. TIARA's TMU runs in parallel with the main data-paths of the system and operates on a set of extra bits tagging each word with data-type, bounds, and security context information. Operations that violate the intended invariants of the system are trapped, while normal results are tagged with information derived from the tags of the input operands. Because of the critical role of tags in enforcing security properties, we call TIARA a *Security Tagged Architecture.*

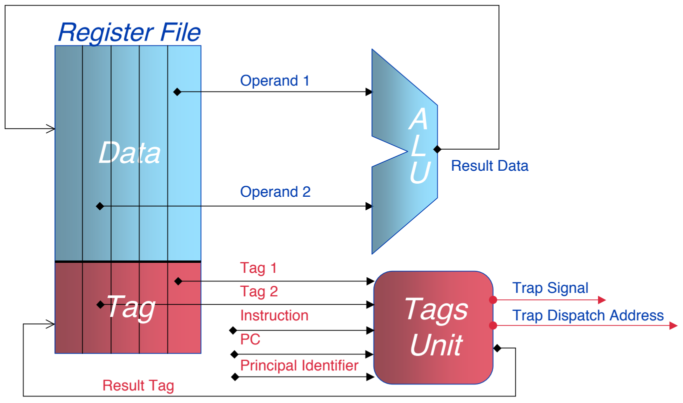


Figure 4: The TIARA Tagged Data Path

Each word in TIARA's memory as well as the contents of each processor register is tagged with a set of extra bits that encode its data-type and its security context. Even the Program Counter (PC) is tagged, allowing the PC to encode the security context of any data that was used in conditional branch instructions. Each process has associated with it a "Principal" representing the current privileges of that process; a processor register holds this value while the process is active.

While TIARA's main data path executes an instruction, the TMU examines the principal register, the tags of both operands, the tag of the PC and the instruction. If executing the instruction would violate any access control policy, then the tag unit causes the process to branch to a "security violation" trap handler; otherwise, the tag unit computes the tag of the result.

Because all words are tagged with their data type, differences between instructions, immediate data (e.g. numbers), and object references are manifest at run-time. TIARA regards all data as objects; in particular, all non-immediate data is accessed through object references that encode both the location and size of the object. All accesses to memory are mediated by object references and all accesses perform bounds-checks in parallel with the load or store, trapping out of bounds references before they take effect.

2.5 TIARA System Layers

The TIARA hardware supports a series of software layers that provide for the enforcement of structuring constraints, access controls and data accountability. The structure of the overall system is shown in Figure 5.

The role of the TIARA hardware is to guarantee integrity of the basic memory structuring conventions: All accesses to memory are mediated by object references and are bounds-checked (so there is no ability to overwrite arbitrary areas of memory); no object to which there are existing references can be deallocated (i.e. there are no dangling pointers). In addition, the hardware is responsible for implementing "secure information flow" policies and these policies are enforced at the granularity of the individual word. Each word has both a data-type tag and a security context tag. These later tags form a lattice [22] and the hardware enforces flow policies with respect to this lattice, tagging every result with its appropriate security context.

This establishes a firm and non-bypassable base upon which several layers of software are constructed; each layer provides increasing guarantees and greater accountability. The first of the software layers establishes a consistent object-oriented level of computing while higher layers establish non-bypassable wrappers, access controls, and data provenance tracking. TIARA includes a novel "plan level" of computing in which code is executed in parallel with an abstract model (or executable specification) of the system that checks whether the code behaves as intended.

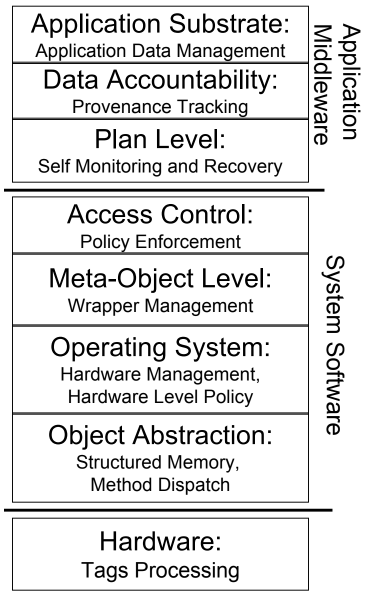


Figure 5: TIARA Layers

The first of the software layers is the *Objection Abstraction Layer* in which memory is structured into distinct objects, each characterized by identity, type and extent. This layer erects a computational model that consists solely of the application of a function to a set of objects. TIARA functions are *generic functions*[33], fully polymorphic functions whose implementation is provided by one or more methods. Function invocation involves dispatching to a specific method based on the types of the operands. The fields of an object may be accessed only by invoking a method on that object and these are subject to hardware enforced access controls. This layer also erects a class system in which every object is a member of some class; classes themselves, being objects, are members of a *meta-class*. The main operations of the object system are described by methods on meta-classes and these are the only means for manipulating the internals of classes.

The *Operating System Layer* controls the hardware and manages the core resources of the TIARA architecture. In spite of the name, TIARA does not have an operating system in the classical sense of a distinguished component, executing in a separate context and possessing unlimited privileges.

Rather the various functions of a traditional operating system (e.g. the scheduler, memory manager, etc.) are implemented by distinct objects with limited scope of capability and privilege. All objects live within a single flat address space; there is no need to separate kernel space from user space since all objects are inherently protected from one another. Each critical component of the operating system layer (e.g. the scheduler, the virtual memory manager, device drivers) has a limited set of responsibilities and an equally limited set of privileges. These components interact according to a set of system-wide global invariants. Thus, TIARA protects its critical resources by adhering to strict enforcement of the object abstraction and to the principle of least privilege. Objects interact using a single mechanism: the function call and all function calls are checked for compliance with access control policies.

The *Meta-Object Layer* is concerned with the imposition of non-bypassable wrappers that are used to provision redundant copies of data, to monitor execution, to track dependencies and to impose access controls. Wrappers are an inherent part of our object model, which is derived from the *Meta-Object Protocol* (MOP) of Common Lisp Object System (CLOS) [33]. A wrapper is simply a distinguished type of method that is combined with other methods in such a way that the wrapper gets control before other methods, allowing it to control whether the other methods are invoked and with what arguments. In addition, the wrapper method gains control after the other methods execute, allowing it to capture and/or modify the returned results.

The *Access Control Layer* is capable of supporting a variety of role-based (and other) access control models using capabilities provided by the lower layers. As mentioned earlier, TIARA's hardware guarantees that every datum is tagged with both its data-type and its security context; the processor checks every instruction to make sure that the executing process has the privileges necessary to execute that instruction on data from the security contexts of the operands.

The access control layer supplements these hardware checks using *access control wrappers* that check whether it is legitimate for the executing process to invoke the indicated function on the operands. It does this by checking the security context and data-type parts of the operands tags. Access control wrappers are imposed using functions provided within the MOP. Since these MOP entry points are themselves just normal functions, access control wrappers may also be imposed on them, allowing us to use the normal access control mechanisms to control who is allowed to impose a wrapper.

Access control wrappers typically are generated from a more abstract description of an access control policy written in an Access Control Language. It was not a major part of our agenda to develop such a language, however we have created a simple one to drive our early work. We anticipate that TIARA can support a broad range of access control policies.

The *Plan Layer* uses system-wide models of the intended behavior to enforce constraints on control and data flows, and to check intended invariant conditions. If the behavior of the system does not correspond to the intended behavior predicted by the system model, then execution is aborted and the dependency records are used to diagnose the cause of misbehavior, to identify data that should not be trusted and to identify what individuals might have been responsible for the failure.

The *Data Accountability layer* is responsible for tracking the provenance of data as directed by the plan layer. Part of this is accomplished by the TIARA hardware which tags all data with its security context. *Data Accountability wrappers* are used to capture the inputs and outputs of all functions of interest and to build dependency records linking the outputs to the inputs and to the invoked function. In addition, data accountability wrappers are interposed around all HCI (Human

Computer Interaction) input and output operations. Output functions track what data has been exposed to which users while wrappers around input routines track which users have contributed data to the computational process.

The *Application Substrate* uses the features of all the lower layers to provide a platform for applications that rely on accountable information flow. It provides a “truth-maintenance” capability that allows conclusions in application documents to be removed if the raw data or computations that led to them are no longer trusted; it also provides a Bayesian reasoning substrate that assesses the trust that should be placed in conclusions given the trustworthiness of the inputs (e.g. sensor data, human judgment, computational tools) to the intelligence analysis process. This substrate also provides tools that hide sections of a document whose security context requires more privileges than are possessed by the audience.

The end result is that application systems are continually checked at runtime to see if they behave as intended, violations of intended invariants are detected and prevented and the provenance of all data is made manifest. Thus, even when attacks succeed, the system knows what data is trustable and what data is suspect. In addition, all data can be traced back to the computational processes, raw data and individuals from which the data was derived. Bayesian inference techniques are used to rate the reliability of conclusions given the trust accorded to the contributing inputs, computational processes and human analysts. Finally since every datum is tagged with its security context, it is easy to identify sensitive data that must be shielded from users without adequate privileges.

These techniques derive from research over the last decade in the Massachusetts Institute of Technology ARIES and AWDRAT projects [14, 13, 67]. In the rest of this report we present more of the details of each of these software layers as well as a description of the proposed core TIARA hardware.

In the TIARA design, we try to unify these various capabilities and to use the lower level facilities to guarantee that facilities higher in the stack are not bypassable. We use TIARA’s STA hardware to guarantee that no attacker can modify or subvert ZKOS and we use both STA and ZKOS to guarantee the non-bypassability of the access controls, reference monitors, and provenance tracking facilities provided at the application layer.

2.6 TIARA Hardware

At a high level, our goal is to track information flows in order to understand the provenance of significant data, to provide confinement [36] and to guarantee that all information flow and access control policies are adhered to. The TIARA hardware contributes to this goal by systematically tracking the security context of all data in the machine, guaranteeing that no process can gain access to data for which it lacks adequate privileges.

The key TIARA hardware structure is shown conceptually in figure 4. The upper half of the picture might be virtually any conventional processor design; it fetches operands from a register file, combines them through an ALU (Arithmetic Logic Unit) in accordance with the current instruction and then writes the results back into the register file. The lower half is the Tag Management Unit; this fetches the tags of the operands from the register file, the identifier of the processes’s current principal from the principal register, the tag of the Program Counter and the current instruction and produces a new tag that that is written back into the tag section of the register file as well as a new tag for the Program Counter. The role of this TMU is to enforce basic structuring conventions, data type consistency and compliance with access control policies while imposing minimal delays.

In addition to the TMU, TIARA provides a few other novel pieces of hardware: A Bounds

Checking Unit (essentially an adder) to guarantee that accesses to objects are within range; and a specialized hardware stack that help in managing the security context of the program counter. We will next describe these, starting with the implementation of the TMU.

2.7 The Object Abstraction

The hardware features outlined above are the primitive building blocks for creating a software abstraction layer in which 1) All of memory is regarded as consisting of objects with definite type, extent and identity and 2) All operations performed on these objects are semantically meaningful and consistent with the object types. Thus, raw pointers to arbitrary memory locations are replaced by "object references". The use of arbitrary operations on raw data is replaced by bounds and type-checked semantic operations on structured objects. In particular, raw pointer arithmetic, buffer overflows and the storing of data in arbitrary locations are impossible. This layer is referred to as the Object Abstraction and it represents the base upon which a variety of software object models (e.g. those of C++, Java, C#, Python, Aspect-J, or Common Lisp) may be constructed.

We will employ an object model that is a generalization of these specific object models and that draws heavily on the ideas in CLOS [33] and Aspect-J [34]. The model provides for a class lattice (i.e. multiple inheritance) and for multi-argument method dispatch (as is done in CLOS). This supports a view that combines functional and object oriented programming: A function dispatches to a method that is consistent with the types of its operands; if there are no applicable methods then the application of the function to its arguments is illegal and traps. The Hash Execution (HEX) unit, described later, efficiently supports method dispatch.

2.8 The System Software Layer

The System Software Layer is responsible for implementing the functionality provided by a conventional operating system such as interrupt handlers, device drivers, trap handlers, management of the physical hardware, resource allocation and scheduling, virtual memory management, persistent storage, authorization and authentication.

In TIARA, each of these is implemented as an individual object with internal storage that is isolated from other components of the operating system layer and that has extremely limited privileges. For example, the scheduler maintains internal information about processor usage and about the priority levels of both system and user processes; this information is inaccessible to other system software components. Conversely, the scheduler has no privileges to access the internals of processes; it cannot for example, read or modify the internal data structures of a process. Device drivers, in particular have extremely limited access rights; they are given a standard object reference with base and bounds that describes the block of memory that is be read or written. They have no other ability to access memory and cannot overwrite storage at random. A major goal of our design efforts will be to determine precisely what module boundaries make sense and what security contexts are needed to enforce the contexts.

2.9 The Wrapper and Meta Control Layer

In addition to these basic features, our object model provides for a "Meta Object Protocol" in which classes are regarded as instances of other classes whose methods implement the basic operations of the base classes (e.g. method dispatch, object creation). This layer also provides for method combination (as in CLOS [33] and Aspect-J [34]); in particular this allows for "wrapper methods" to be applied to base methods to implement the concerns of a distinct aspect such as access control or data provenance tracing. Wrapper methods execute before and after the base method and control whether the base method is executed at all. This allows us to construct a series of more abstract software layers that provide for access control, flow monitoring and logging and for self-checking and diagnosis.

Since classes are themselves objects, their behavior is described by another set of classes, commonly termed “Meta-Classes”. Methods on these classes control how normal classes behave; in particular they describe how method combination and method dispatch is implemented. This allows us to distinguish different classes of wrappers, such as those used by TIARA internally to implement access and flow controls, from normal user level wrappers. In addition, the meta-level allows us to control who is allowed to impose such wrappers, making the imposition of security-oriented wrappers non-bypassable.

Wrappers are available as part of the toolkit of the application programmer; however, there are a distinguished set of meta-level wrappers that are imposed by system software to implement access controls and to track data provenance. Such wrappers are non-bypassable and access controls imposed at the meta-object level restrict the use of such wrappers to processes executing with special privileges. Since these access controls are ultimately enforced by the hardware, this allows us to create a very strong base for security and provenance management.

2.10 The Access Control Layer

Using the object abstraction as a base, we can provide for the efficient enforcement of a variety of access control policies. The outline of this is as follows: We associate with each process a stack of “Principals”, where a principal represents the privileges extended to the current process; the process may dynamically (and temporarily) assume a new principal, but it must authenticate itself to do so. All operations on data are ultimately performed by dispatching to a method based on the data type and security contexts of the operands. TIARA extends this notion to include the principal in the method dispatch. Thus a method is dispatched to only if the privileges of the Principal are consistent with the data types of the operands; otherwise an illegal operation is signaled. Role-based access control systems (and a variety of other access control schemes) are easily mapped into this framework: The role is encoded in the Principal; The type hierarchy is encoded in the object-type lattice; The operation type is encoded in the method-name. In our model, the MOP and its use of meta-classes and wrapper methods allows us to simultaneously implement a variety of access control policies. Furthermore, these are all implemented through method dispatching and this is efficiently supported by the HEX hardware.

In many ways, TIARA is similar to a capability architecture [47, 37, 32, 66]. However there is an important distinction. Capability systems mix together an object reference with the set of access rights; thus, when a process transmits a capability from one component to another, it transmits not only knowledge of the referenced component, but also a set of rights to access that component in particular ways. This leads to problems with managing revocation of privileges and with controlling the amount of privilege that is delegated. Although there are ways of dealing with these issues in appropriately structured object oriented capability systems [42], in our view these impose unneeded complexity.

TIARA separates knowledge of an object (which is carried in a normal object reference) from rights to access that object (which is represented in the Principal Register of a process). One can think of this in terms of Lampson’s notion of an access control matrix [35], which uses one dimension to represent Principals and the other to represent Objects. Each cell of the matrix represents the access rights of a principal and to an object. Capability systems cut this matrix up into columns while Access-control lists cut it up into rows. TIARA, instead, operates on the matrix directly, since it represents the principal separately from the objects that are being accessed.

2.11 The Plan Layer

The Plan Layer is responsible for monitoring whether a process is behaving as intended; the design of this layer derives from our work on the AWDRAT [67] system within the Defense Advanced Research Projects Agency (DARPA) Self-Regenerative Systems (SRS) program. This layer is driven by an abstract model, or Plan, of the intended computation; this consists of a hierarchical block diagram of the computation, annotated with data- and control-flow links between the components and Pre- and Post-conditions around each component. This abstract description drives the synthesis of wrapper methods that are used to gather information on entry and exit from the relevant computational components and to check that this information is consistent with the constraints of the Plan. The previous layers check that every operation is permitted and semantically meaningful within its local context; this layer checks that the global constraints of the computation are respected. In addition, the plan layer is responsible for capturing and provisioning redundant copies of data to support recovery from attacks. This backup data is used by the plan layer, together with its model of the computational process, to facilitate the reconstitution of a consistent state of the computation from which the computation may proceed.

2.12 The Data Provenance Layer

The Object layer guarantees that only semantically sensible operations may be performed; the Access Control layer strengthens the guarantee to rule out operations that are inconsistent with access policies.

The data provenance layer provides for accountability by tracking how the current state of each object is dependent on the state of other objects. The HEX hardware performs the most basic part of this operation since it performs fine-grained "tainting" of data (e.g. it maintains the security context for each individual word). At a higher level, the data provenance layer builds links that connect an object to those other objects that were used in computing its current state. These links represent dependencies and the dependencies form a network similar to those of "Truth Maintenance Systems" (TMS) and Bayesian networks used in Artificial Intelligence.

These capabilities, like those of the Access Control layer, are implemented by wrapper methods that work in tandem with the hardware's HEX units.

2.13 The Application Substrate Level

The Application substrate level provides a variety of services that are supported by the dependency network built by the data provenance layer. As mentioned in section 2.12 this dependency network is similar to the data structures of a TMS. One of the key capabilities provided by a TMS is "assumption retraction" which invalidates all statements that depend on an abandoned assumption. The equivalent operation for the application substrate layer is performed when some datum is identified as having been corrupted by an attack; at this point the Data Provenance dependencies are traced to identify all data whose current state was derived from the corrupted datum and to mark these data as untrusted.

The dependency network is also similar to the data structures of Bayesian networks and it is used for probabilistic reasoning. The Data Provenance layer dependency network can also be used to support such reasoning: Each raw observation, whether it comes from sensor data or from human sources, is accorded a degree of trust, represented as a probability. For each step in the processing of intelligence data, the data provenance layer builds a link in its dependency network which can be interpreted as stating how the trustworthiness of the conclusion depends on the trustworthiness of the inputs as well as on the reliability of the interpretation step itself. Bayesian inference algorithms in the application substrate layer estimate the trustworthiness of the final conclusions (including competing conclusions) and update these estimates when the estimated reliability of any input changes.

3 Related Work

Many of the base observations here (*e.g.*, the vulnerability of large kernels, desirability of formally establishing OS, need to align hardware, OS semantics, and verification) have a long history. The Multics Kernel Project identified how the Multics kernel might be reduced to tens of thousands of lines of code with the promise that future kernels might be formally validated [64]. PSOS developed the design for a provably secure operating system that also exploited tagged data [25, 49]. LOCK targeted a small security kernel with a hardware co-processor to enforce security and judicious use of formal validation [62, 69, 63]. These systems made the necessary compromises to coarse-grained control (*e.g.*, pages and files), to the hardware of the day, and to the available verification technology.

The trend to microkernels (*e.g.*, [27]) embraced the idea that it was beneficial to reduce the kernel, separating out services as separate processes isolated from each other. Performance concerns with commodity hardware kept the separate components relatively large. Even with this concession, performance cost limited adoption. More recently Tanenbaum [72] notes the security virtues of microkernels and suggests the modern importance of security makes it worthwhile to sacrifice performance for security.

The need for strong isolation has been recognized for embedded systems, leading to work on separation kernels (including the work on MILS [56, 1, 50, 21]) which has largely focused on software isolation, with the usual performance considerations discouraging fine-grained compartmentalization. Our work can be seen as allowing these philosophies to be carried to the extreme by supporting them with appropriate fine-grained hardware isolation.

Bell and LaPadula defined a formal security model for Mandatory-Access Control (MAC), including details of the implementation for Multics [10]. Bell notes that hardware capability and performance limitations of the machines forced their model and rules to only work with coarse-grain entities (*e.g.*, files), outlawing many finer-grained interactions which would be consistent with conceptual policy intent, but could not properly be enforced by the hardware of the day [11]. SELinux [29] shows how MAC could be added to Linux, but lacks the more formal basis of MAC in Multics and suffers from many of the underlying weaknesses of the Unix OS.

Security properties concerning information flow have been typically formalized as *noninterference*: see [57] for a survey of the research on noninterference and [48] for a Java-based programming language with a type system that supports information flow control based on noninterference. In practice, the noninterefereence requirement is too strong a requirement; see [58] for a survey of methods for defining such relaxations via declassification. Program analysis for (variants of) noninterference has been examined in literature. The approaches that have been considered include slicing [70] or using a logic for information flow [3]. Our metadata tags can carry taint information; this allows dynamic tainting in cases where analysis cannot establish static taint labels and supports the dynamic taint propagation with hardware so there is no added, runtime overhead for this dynamic taint propagation.

Previous machines have employed tagged data, including Burroughs-Unisys MCP/AS [52, 30], the Intel 432 [53, 31], multiple generations of the Symbolics LISP Machine [45, 6], IBM's System 38 [38] and the Cambridge CAP machine [76]. Many of these machines employed a hardware/software architecture whereby access control to objects required possession of a hardware *capability* to perform the controlled action. A main advantage of the capability concept is in enforcing least-privilege, since capability-granting is analogous to "need to know."

The Burroughs machine used tags for security, but security guarantees required that all code be written in a high-level language and compiled by their compiler, a requirement that was not otherwise enforced by the system. System 38 gained considerable protection benefits from a single tag bit. The LISP Machine used tagging primarily for efficient support of LISP and graceful detection and handling of runtime errors (*e.g.* unintentionally treating a floating-point value as a pointer); this use does offer a practical improvement to security, minimizing the likelihood a latent programming bug can be exploited to create a breach, although security and isolation were not design goals of the LISP Machine processors or OS. In contrast, our STA design exploits the lower relative cost of hardware today to support a large metadata space and an implementation capable of providing rigorous runtime security checking for all programs. This enables a new operating system model, ZKOS, and close coupling with modern formal methods to reallocate roles in the design tradespace.

Recently, several efforts have begun to explore the use of metadata tags for security (*e.g.*, [77, 71, 73, 17, 19, 4, 75, 78, 23]). These results illustrate the promise of the technique but generally lack the grounding to make strong security guarantees. Prior work does not use the hardware tags to enable OS redesign, does not formally ground the protection provided by the tagging, and does not exploit the metadata tagging to enable software verification. We take these ideas further by systematically identifying the semantic invariants, carefully designing the tagging along with the OS and verification strategy [40, 20]. Prior work solutions for cooperation of compilers and hardware support (*e.g.*, [73]) and efficient microarchitectures that reduce tag memory overhead (*e.g.*, [23, 77]) can be useful in minimizing the overhead required for an implementation.

Virtualization and hypervisors [18, 55, 8] have received renewed attention as a protection solution, in addition to their use for resource-sharing. Computations are protected from each other at a very coarse granularity (the Virtual Machine, or VM) by a complex software system, which may or may not be OS-aware. As their "unit" of protection is the VM image, such systems alone do not offer the rich fine-grained control and natural ties to formal methods that our approach provides.

4 Zero Kernel Operating Systems (ZKOS)

In the rest of this document we will examine the details of the TIARA design, aggregating these into three major sections: The first will be concerned with system software, the second with the hardware architecture and the third with application infrastructure. It is difficult to segregate the discussion into these three areas because the overall project involves a co-design of all three areas.

Our goal is to design a system in which we can place high trust; ultimately we would like to formally prove that this trust is warranted. As we argued in Section 2.1, the source of the problem is that our systems do not maintain enough metadata to actively enforce the intended semantics of the computation. In the TIARA design, however, we have the opportunity to add features to the STA hardware architecture that make the intended semantics manifest and therefore make it easier to prove that the overall system behaves as intended.

As we noted in Section 2, Saltzer and Shroeder [59] identified several key principles for system design, including:

- *Complete mediation*: Every access to every object must be checked for authority.
- *Least privilege*: Each module is granted only the minimal privileges necessary to do its job.
- *Separation of privilege*: Protection mechanisms should require that more than one condition should be met before access is permitted.

The rest of this document will present an examination of the ways in which a co-design of the system software, the hardware architecture and the application infrastructure can meet these criteria. We begin our discussion with the operating system. Then we will look at the details of the hardware design and then finally we will turn to how the application infrastructure can capitalize on the facilities provided by both the hardware and system software.

We begin with the ZKOS operating system, because the construction of an operating system embodying these three principles will motivate most of the hardware architecture design. This will include both features of very general applicability (e.g. compartments, principals, access rules) as well as hardware features designed to solve specific problems. As we will see in Section 4.4, global garbage collection presents unique problems for a system based on the notion of least privilege; however, these problems can be addressed by provisioning new hardware data types, special purpose compartments and novel instructions.

In looking at the ZKOS design we will first examine its general software architecture and the design patterns used in its implementation. We will then turn to several specific elements of the ZKOS design: the process manager, the garbage collector, and the design of Input/Output (I/O) interfaces. We consider these areas in detail because these are areas of higher risk; by focusing on the issues raised in these areas, we believe that we will address issues raised in other areas of system software.

4.1 The ZKOS Software Architecture

One goal of any operating system is to protect the "system" code from "user" software and to segregate the information of one user from that of another. Virtual memory has been the only hardware tool available to system software designers; it is used to create a separate virtual address space for each user process. In addition, there is a special "kernel" mode to support system software. Software running in kernel mode has unlimited privileges and can both read and write any word of memory. As it turns out, switching context between user and kernel mode is a very expensive operation, involving thousands of cycles. Faced with this cost, operating system engineers have designed their systems to avoid crossing the user-kernel barrier as much as possible. In practice, this has meant moving more and more software into the kernel, even when there is no logical need for that software to have unlimited privilege.

We can see in this two related problems: First, switching level of privilege is expensive; Second, there are only two levels of privilege. Some hardware, going back to [65], provide several levels of privilege, arranged in a hierarchy of increasing privilege. While this is an improvement, it still makes it difficult to make fine grained distinctions about the specific privileges any system software component actually needs to do its job. Far too many elements of the system end up with far too many privileges and with the ability to further escalate those privileges with ease. This, in turn, leads to a design in which a single vulnerability can be exploited to gain unlimited access to system and user resources.

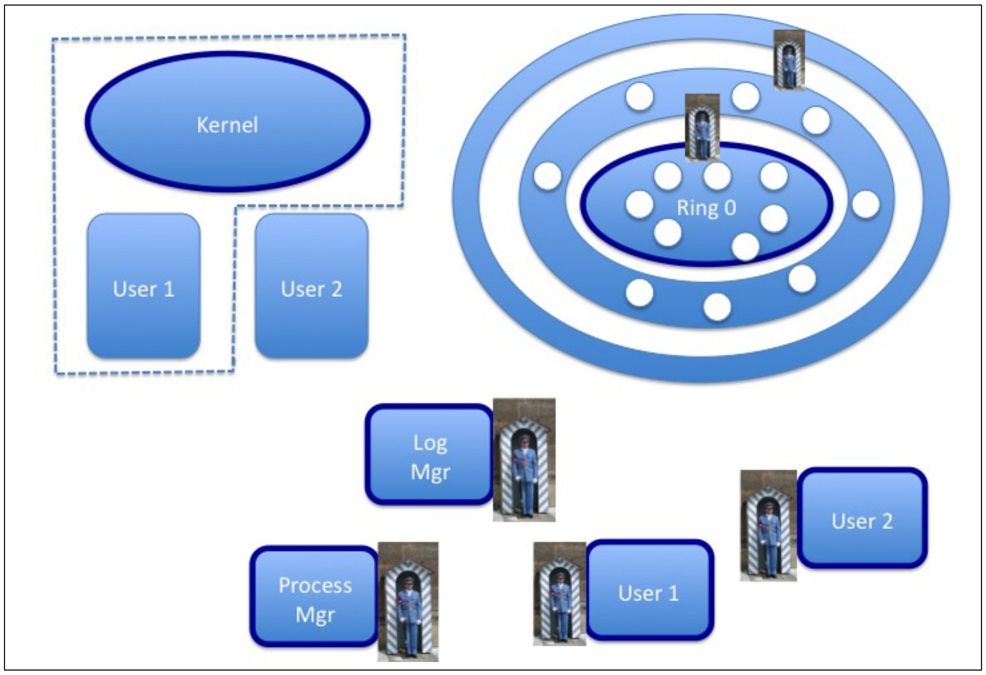


Figure 6: Single Kernel, Ring and Compartmented OS Models

The alternative, which we explore in the design of ZKOS, is not to rely on the memory barrier at all, but rather to use the STA hardware's notion of compartment and principal in order to decompose the operating system into a large number of small components each of which has its own compartment in which it stores its own data. Each component has its own principal under whose authority it operates. To first order, a component's principal has exclusive access to its own data and lacks the privilege to access data of other system components or of user data. When such access is required, it must be provided for through the explicit use of shared compartments. Figure 6 illustrates the differences between the three styles of operating system designs.

It is worth noticing, before going into further details, that a compartmentalized system like ZKOS actually makes no fundamental distinction between operating system and other code. All code is broken down into modules that have access to data in only a limited number of compartments. There is no memory barrier or kernel/user distinction. As a result, whatever structuring principles are used to guarantee information flow properties of ZKOS can be applied equally well to any other layer of the software system.

If we consider how application middleware, (e.g. a web-server) is treated in conventional systems, we can see the clear advantage of the ZKOS approach. A typical web-server implementation manages a set of threads, one for each client request; however, because memory barrier protection incurs a huge context switching overhead, web servers intermingle the data of all their clients within a single address space, foregoing the use of the only protection mechanism afforded by the operating system and hardware. In ZKOS, the clients of a web server can operate within a common address space, while still having a strong guarantee of information isolation provided by STA compartments. In effect, the web server is really another layer of operating system. In ZKOS it can be built in exactly the same way as the OS and can reap the same protections; in a conventional system, it is a distinct from the OS (or inappropriately included in the OS) and has far fewer guarantees available to it.

4.1.1 STA Provided Building Blocks

The STA hardware provides a set of hardware supported building blocks with which we can construct the ZKOS software architecture. These include:

- Data Types: Every word of memory is tagged with its data type. STA provides for a large space of data types, including the common types (e.g. integer, float, instruction) as well as a large space of special purpose data types (e.g. Principal, Compartment, Garbage-collector scan pointer) that are related to the STA hardware architecture.
- Compartments: Every word of memory is tagged with its "compartment", an aggregation of objects upon which identical access controls are imposed.
- Principals: A representation of the entity on whose behalf a thread of control executes.
- Access Rules: These are applied on every instruction and regulate which principals are allowed to perform which instructions on data in which compartments. These are stated in a language drawing ideas from Domain and Type Enforcement [5] and Role Based Access Control [9, 26, 60] as is done in SELinux [29].
- Gates: These are compound objects, including a procedure, a compartment and a principal. When a gate is invoked, the principal and compartment are rebound to those in the gate and the procedure is then invoked. When the procedure returns, the principal and compartment are restored to those in effect prior to the invocation of the gate.

The STA hardware architecture provides a large tag, capable of referencing a very large space of compartments for structuring memory, while the processor principal register is capable of referencing a very large number of principals. Thus, we are free to design the system as if there is an unlimited number of compartments and principals and to use these to make fine grained distinctions.

Conceptually, there are several types of access rules each related to the category of instructions to which they apply:

- Instruction compartment rules: Instructions, like other words in memory, belong to a compartment. The first category of access rules controls whether a principal is allowed to execute code within the compartment. (Instructions themselves are contained within a procedure object, which might have a different compartment from that of the instructions within it; in addition, instructions within a procedure might be in several separate compartments; we haven't yet explored whether this offers useful opportunities for enforcing privilege containment).

- Procedure Call operations: Procedure call instructions take a reference to a procedure (or a gate) as well as some ancillary information (see Section 5.7). Procedure-Call access rules consider the running principal and the compartment of the procedure being invoked. The rule specifies whether the procedure invocation is permitted.

- Basic operations: These rules apply to typical 2 and 3 operand instructions (e.g. Add, Multiply). They specify whether the operation is legitimate (e.g. Integer add must take 2 integer operands) and if so what data-type and compartment the result should have.

- Load operations: These rules encode who is allowed to read data in which compartments. All load operations take a base register and offset, and load data into a destination register. The base register should point to an object in memory and the offset indicates which slot of the object is to be fetched. Load access rules consider the running principal, the compartment of referenced object and the compartment of the word being fetched from the referenced slot of the object. The rule specifies whether the load is allowed and if so, what compartment should be assigned to the loaded word.

- Store operations: These rules encode who is allowed to write data into which compartments. Similar to the Load instructions, Store instructions take a base register and offset; the third argument is the write data. Store access rules consider the running principal, the compartment of the referenced object, the compartment of the data in the slot being overwritten and the compartment of the write data. The rule specifies whether the store is allowed and if so what compartment should be assigned to the stored word.

- Special purpose instruction: The STA hardware provides for a number of special purpose data types and instructions in order to allow control over system resources at an appropriate level of granularity. For example, in the design of the garbage collector (GC), we consider both data-types and instructions that are intended to be used only by the GC. In the Process-manager, we provide data types for special data structures holding the processor state and allow these only to be manipulated by special instructions who use is limited to the Process-manager principal. Special purpose instructions may be implemented in the micro-architecture of an STA implementation by trapping to software handlers or they may be directly implemented in hardware (or microcode). Special purpose instruction rules control what principals are allowed to access the specific resources manipulated by these special purpose instructions; these consider the data type and compartments of the operands as well as the running principal and the specific special purpose instruction.

4.1.2 Using The Building Blocks to Structure ZKOS

This set of building blocks allow us to radically modularize ZKOS, assigning to each software component its own private compartment(s) for storing data and code. In addition, each component has its own principal(s) and a set of access rules that govern the rights other components have to access the code and data in its compartments. Each component is granted only the *least privilege*

[59] necessary to do its job. Usually each OS component has no rights to access data in the compartments of other OS components (or in user components) and this is enforced by the hardware Tag Management Unit. Because the TMU hardware enforces separation between its components, ZKOS has no need of a virtual memory boundary; instead we allow all software to live in a common flat address space and provide for global, generational garbage collection [43, 7, 24] as described in Section 4.4. We note that virtual memory hardware is still provided and can be used when appropriate to support isolation of Direct Memory Addressing (DMA) devices or the garbage collector if desirable.

The core of each ZKOS component is implemented as a set of procedures accessing data within a common compartment and running under the authority of a single principal. The procedures (and the instructions within these procedures) of the component are stored in another compartment reserved for the component's code. The only principal authorized to run these procedures is that of the component. This is encoded in procedure and instruction access rules that are enforced by the STA. This technique is used to guarantee isolation among components.

However, ZKOS components also need to interact and to share information; we want these interactions to be structured and the sharing of information to be controlled so that no component is granted privileges beyond those needed to perform its job. To allow components to interact in such a structured manner, each component establishes a set of "gates" (*i.e.*, packages of a procedure, a principal and a compartment see Section 5) representing service entry points. Gates are the sole mechanism through which a thread can change privilege (this is guided by another principle stated in [59], the principle of economy of mechanism).

The right to invoke a gate is extended (via access rules) only to those other OS (or user) components that need to invoke the service provided by the gate; this too is enforced by the STA's TMU. Typically associated with this gate is a separate compartment and principal. The compartment is used to hold data shared between the two components. The principal can be thought of as a "proxy principal" having some of the privileges of both components, but typically, it is less privileged than either component's core principal. As the name implies, this principal can be thought of as a "proxy"; for example, if the Log-Manager creates a gate to mediate service calls from the Process-Manager, then the principal in the gate can be thought of as the "Log-Manager acting on behalf of the Process-Manager". This principal need not have access to the core data of the Log-Manager; rather it has access to data stored in the shared compartment. The proxy itself, has access to other gates that allow it to temporarily run as the Log-Manager; but note that while doing so it does not have access to the data in the shared compartment.

This arrangement allows ZKOS to carefully control what information can flow between components. In addition, it decomposes privileges, granting only a limited number to any component. Thus, a penetration into one component, need not lead to privilege escalation, both because privilege is not organized in a hierarchy and because gaining specific privileges involves penetrating the specific component owning that privilege.

Just as with interactions among ZKOS components, a ZKOS component (*e.g.*, a Log-manager) typically needs limited access to user data (or *vice versa*). These types of limited exchanges are also effected by creating an additional principal to act as a proxy for the OS component in its interaction with user; an additional compartment is used to hold the shared information. These proxy principals and shared compartments are subject to a set of access rules that severely limit the information flows to just those desired; neither the OS component nor the user gains unintended privileges. As shown in Figure 7, the user interacts with the OS component (*e.g.*, the log manager)

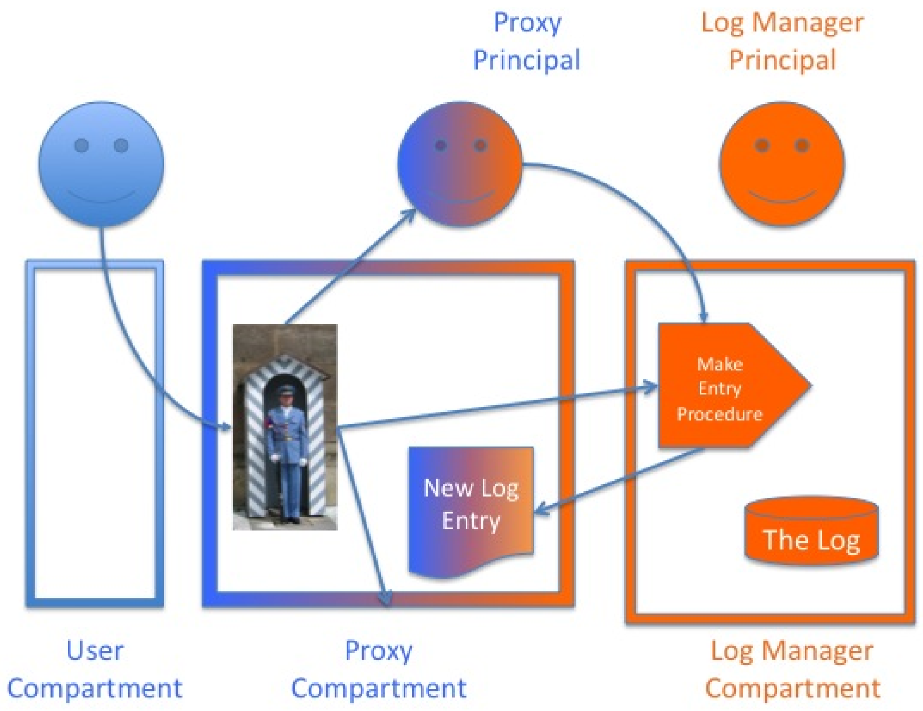


Figure 7: Components Interact via Proxy Compartment and Principals

by invoking a gate in a compartment shared by the user and log manager; the gate builds a new log entry in the shared compartment while temporarily executing as the log manager's proxy.

While acting as the proxy, the thread is only able to access data in the shared compartment; it is not able to access information in the user's compartment nor in the log manager's compartment. The log manager itself is then invoked via a gate accessible to the proxy principal (but not the user's principal) in order to actually add the entry to the log which is accessible only to the log manager's principal. This strictly limits the flow of information as well as the sequence of principal changes.

The ZKOS design decomposes each OS component into its smallest meaningful pieces, treating each of these, in turn, as a sub-component and assigning unique compartments, principals and access rules to each. In the end, each component is surrounded by a set of compartments and associated principals and gates that allow interactions with other ZKOS components and with user processes. Thus, each component resembles a planet surrounded by a set of satellites and we often use the term "hub and satellite" to refer to this design pattern.

We believe that the hub and satellite and similar patterns are amenable to formal analysis; furthermore, we believe that this formal analysis is much simpler than today's standard verification techniques. In particular, to analyze what information flows are allowed (including what information can be exported, modified or corrupted) one need only analyze the structure of compartments, principals and access rules. We construct a formally analyzable information-flow graph whose nodes are compartments and principals and whose arcs are (1) read access rules linking compartments to principals and (2) write access rules linking principals to compartments.[5] A similar graph, in which the nodes are principals and the arcs are gates, encodes the possible transitions of privilege.

[5]External resources (*e.g.*, files, networks, etc.) are also represented as compartments.

Our goal is then to prove that overall information flow policies are a consequence of the structure of these graphs, in particular of the reachability relationships within the graph structure. Reachability in these graphs sets an upper bound on the possible information flows in the system. If these flows are consistent with the policies one wants to enforce then one need never examine the code to formally guarantee the policies since the code cannot effect any information flow not sanctioned by the access rules. If we cannot establish the desired verification results, then one option is to further modularize the system, creating more compartments, principals, and access rules to explicitly represent distinctions of privilege and limitations of information and control flows.

ZKOS, supported by the STA hardware, in contrast to kernel-oriented systems, provides OS components a strong guarantee not only that their integrity cannot be compromised by user code, but also that their integrity cannot be compromised by other OS components. These guarantees are symmetric, application middleware and user code are similarly guaranteed that compromised system code cannot compromise their integrity, because system components lack the privilege to access user data and code.

This is part of the overall "defense in depth" strategy of ZKOS. The hardware provides grounding for the fundamental integrity of the system. It also provides strong controls on information flow by propagating the compartment labels of all data and restricting the flow of information between compartments (this draws on a common philosophy shared with [78, 48, 74] among others).

Components are also designed to be mutually suspicious, checking the arguments and return values passed between them for reasonableness and subjecting the overall information and control flow to checks by a reference monitor that is driven by an architectural model of the system [68, 67], (see Section 7.2).

In summary, ZKOS components are decomposed into the smallest meaningful units whose interactions and privileges are highly constrained. The information flows among ZKOS components are governed by a set of declarative access rules that are open to formal analysis. Moreover, the tools used to structure the OS (*e.g.*, compartments, principals, access rules and gates) are equally available to application middleware and user software. In all cases, the hardware guarantees that the software cannot violate the access rules.

4.2 An Example of the ZKOS Architecture

In the next several sections we will delve into increasing details of the ZKOS architecture and implementation. In this section we will look at how a few key ZKOS components (e.g. the Log-manager, Login-manager, Process-manager) interact with one another and with user processes. We will also examine aspects of how the system bootstraps itself. We will then turn to a more detailed look at three key ZKOS components: The Process-manager, the Garbage Collector, and the I/O interfaces. As mentioned earlier, we chose these three components because they seemingly require global access to information and therefore pose the most likely source of vulnerabilities leading to unintended information flows. By focusing on these components, we feel that we can show that even the areas of highest risk are amenable to designs satisfying the criteria of least privilege, complete mediation, and separation of privilege.

Using the techniques describe in Section 7.3.1 we have implemented several core components of ZKOS. Each of these follows the hub and satellite pattern: As illustrated in Figure 8, there is a core compartment and principal for each component surrounded by compartments and principals for each entity with which the component interacts. Gates are used to allow clients of the core components to change (not necessarily escalate) their privileges in order to interact with the component.

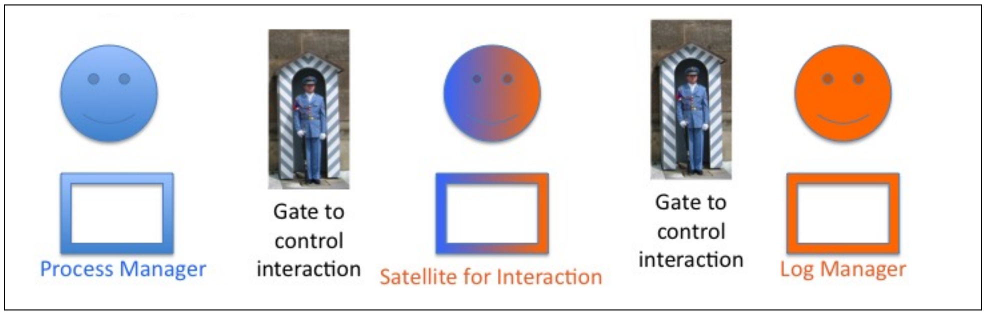

Process Manager
Gate to control interaction
Satellite for Interaction
Gate to control interaction
Log Manager

Figure 8: Components Interact via Shared Compartments, Proxy Principals and Gates

4.2.1 The Log Manager

The Log-manager offers a good example of the issues that need to be addressed. The Log-manager collects data (log entries) from every ZKOS component as well as from every user process; it also maintains a core data structure (the log) which must be kept from serving as a vehicle for unintended information flows (e.g. one user's log entries being read by another user).

The task of the Log-manager is to accept a stream of events signaled by ZKOS components, user code, and by the ZKOS application infrastructure described in Section 7 (these manage access control, execution monitoring and information flow tracking). Events are turned into log records describing what happened as well as the dependencies between events; each record is annotated with a time-stamp as well as the thread that signaled the event and the current principal of the signaling thread. The Log-manager builds log-entries, adds them to the log, commits the records to persistent storage and makes the records available for information retrieval and browsing as shown in Figure 9.

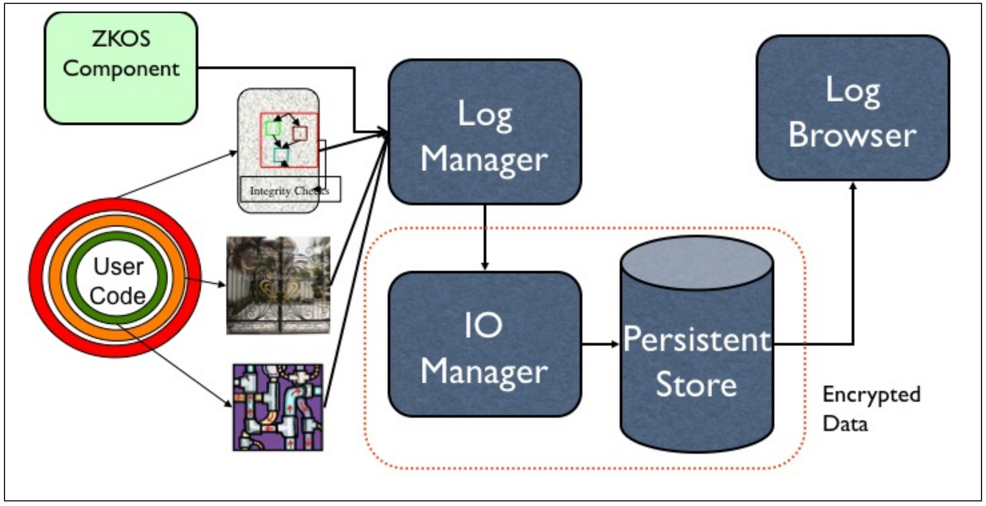


Figure 9: The Log-Manager

The Log-manager accepts information from a variety of sources, both user and ZKOS components, and it makes this information available to a variety of clients including users, system administrators, and forensic investigators (e.g. law enforcement, security specialists). The concern is that, without careful design, the Log-manager could easily become a vehicle for unintended information flow. We do not want the Log-manager to have unlimited access to user data (as it would in a kernel based system) nor do we want users to have unintended access to information controlled by the Log-manager.

To address these concerns, we decompose the overall log system into a set of sub-components: The core component that collects log entries from clients (the log builder), the core component that browses log records (the browser), and the satellite record-builder components used to collect event data from each client. We will not focus particularly on the log browser, except to note that preserving the provenance of each log record and the dependencies between them allows us to build a flexible set of access rules, managing the tradeoffs between the system administrators' needs to gather forensic and management information and user's desire to maintain privacy.

We now analyze what privileges are necessary for each component to do its job. (Consider Figure 10 which illustrates the interaction between the user Login-manager and the Log-Manager). First, we note that the log-builder component has no need to actually read the records. It's job is simply to add the records to the log data-structures and then to commit the records to persistent storage; this second step is accomplished by invoking the I/O system that will be described in Section 4.5. Thus, the log-builder only needs the ability to write a log record into the log data structure.

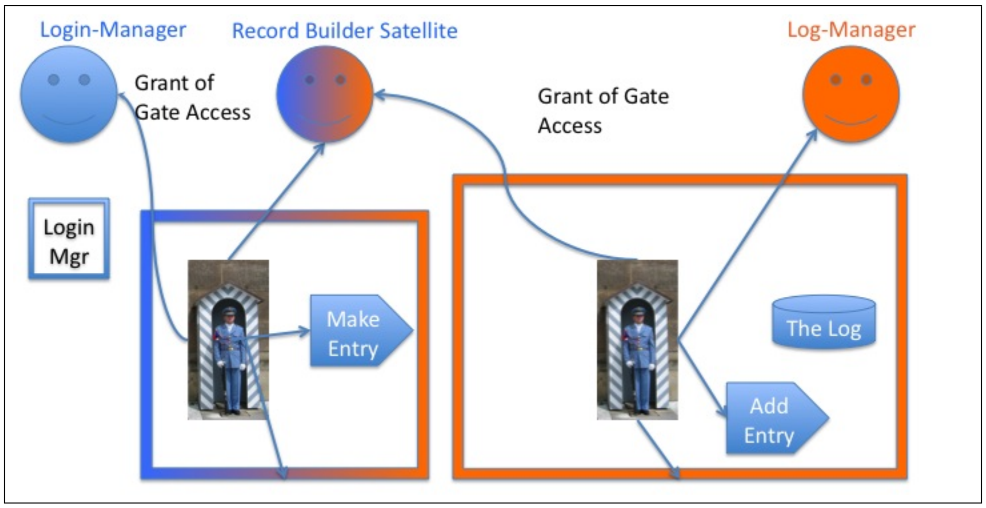


Figure 10: User Manager Interacts with Log-Manager Via Satellites

Now let us consider the issue of building a log record. The "record-builder" that does this needs the ability to create a log record and to add the client's event data to this record. It has no need to read other client data. We, therefore, create a satellite compartment and principal for the task of creating the log record and filling in the user's event data. This record-builder satellite is accessed through a gate (for the Make-Entry procedure) to which only the client has call permission and to which the event data is passed as arguments. The gate changes principal and compartment to that of the satellite record-builder; only its principal can allocate or modify data in the satellite's compartment. The record-builder satellite allocates the record, and fills in the data provided by the client (as arguments passed to the gate) as well as the timestamp, thread and principal; the management of this data is reserved for the record-builder in order to avoid the possibility of the client spoofing the log system. The record-builder satellite, however, cannot actually add the record to the log, because it lacks write access to the log. However, the satellite is provided with a gate (for the Add-Entry procedure) to which it has call access; this gate changes principal and compartment to that of the "log-builder". The satellite invokes this gate, passing it the new log record as an argument.

Analysis of this architecture shows that the client of the Log-manager has no ability to modify the log (other than to request the addition of a new record), the Log-manager cannot access data other than that provided to it through the request to create a new record. Finally, the only thing the Log-manager can do with the data passed to it is to create a record and add it to the log; no other information flow is allowed.

4.2.2 Bootstrapping ZKOS Components

The brief discussion of the log-manager above illustrates how ZKOS manages information flow by 1) Decomposing its components into smaller components with limited capabilities and limited requirements for information access and 2) Granting to each of these the least privileges actually required. The structure of compartments and principals is non-hierarchical; each principal plays a particular role and has only those privileges it actually needs. The flow through a particular service consists of a series of transitions or privileges mediated by gates.

This architecture requires the various ZKOS components to construct a set of satellites consisting of principals, compartments and gates that allows interacting components to communicate. In addition, those ZKOS components that provide services directly to user processes must also construct satellites dynamically linking themselves to the users' processes as users log into the system. To see how this works, we'll consider how a ZKOS component such as the Log-Manager initiates its interactions with the user Login-Manager.

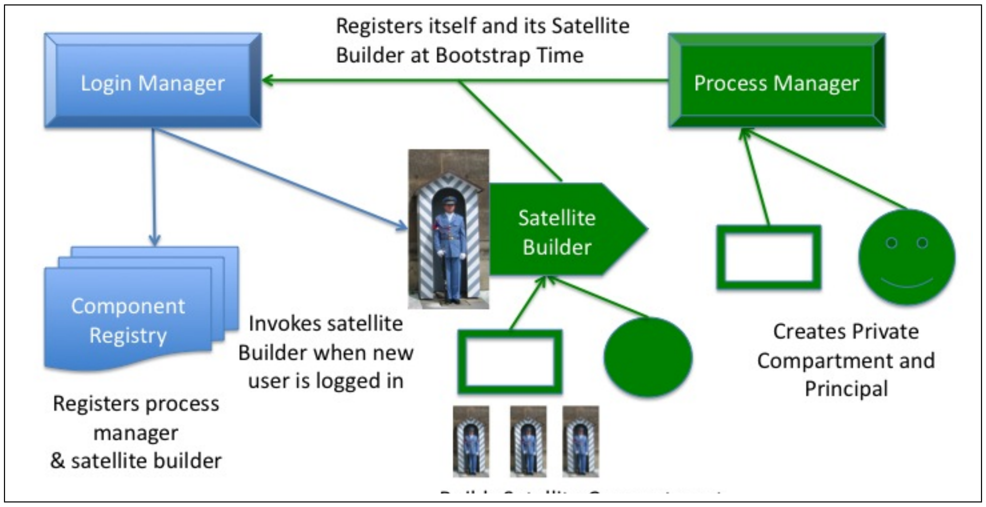

Registers itself and its Satellite Builder at Bootstrap Time
Login Manager
Process Manager
Satellite Builder
Component Registry
Invokes satellite Builder when new user is logged in
Creates Private Compartment and Principal
Registers process manager & satellite builder

Figure 11: A Component Registers Itself with The User Manager

The simpler part of the bootstrap involves creating the satellites linking ZKOS components. To do this, each component creates a compartment and principal for each of the other ZKOS components to which it wishes to provide services. Each entry point of the component is then packaged as a gate which is registered with the client component during a linkage step performed after all components are loaded. For example, the Log-Manager creates a gate for the Make-Entry service and this is passed to the Login-manager (so that new user log in events can be logged). To enable the registration, each component (e.g. the Login-manager) provides a "register-service" entry point (i.e. a gate that adds the Log-manager's entry-point to the Login-manager's service registry after shifting to the Login-manager's principal and compartment). All ZKOS components are provided call access to the register-service gate.

The process for dynamically linking user processes to ZKOS services is a bit more complex. Unlike the linkage between system components, this linkage must be performed whenever a new user process is created. Consider how a new process would be enabled to make log entries. This would require the Log-manager to create appropriate principals, compartments, and gates for the new user process when the user process is launched. At system bootstrap time, the Log-manager creates a gate which it registers with the Login-manager. This is not the gate discussed above, but rather another one that will be called later whenever a new user process is created; it will be passed this new process as an argument. This is registered in a second registry managed by the Login-manager as illustrated in Figure 11.

When a new user is logged in and its process is launched (through interaction with the Process-manager), the Login-Manager consults this second registry, invoking each of the registered gates in turn. Each of these gates transitions to the principal of its ZKOS component and then creates a new satellite structure linking the new user process to the ZKOS component. This satellite structure is passed back to the Login-Manager and bound in the global environment of the user process (through interaction with the Process-manager). At the end of this process, the new user has a set of service entry points to each of the relevant ZKOS components. Each of these entry points is mediated by a gate that transitions into a satellite compartment and principal private to that user and linking the user process to the ZKOS component.

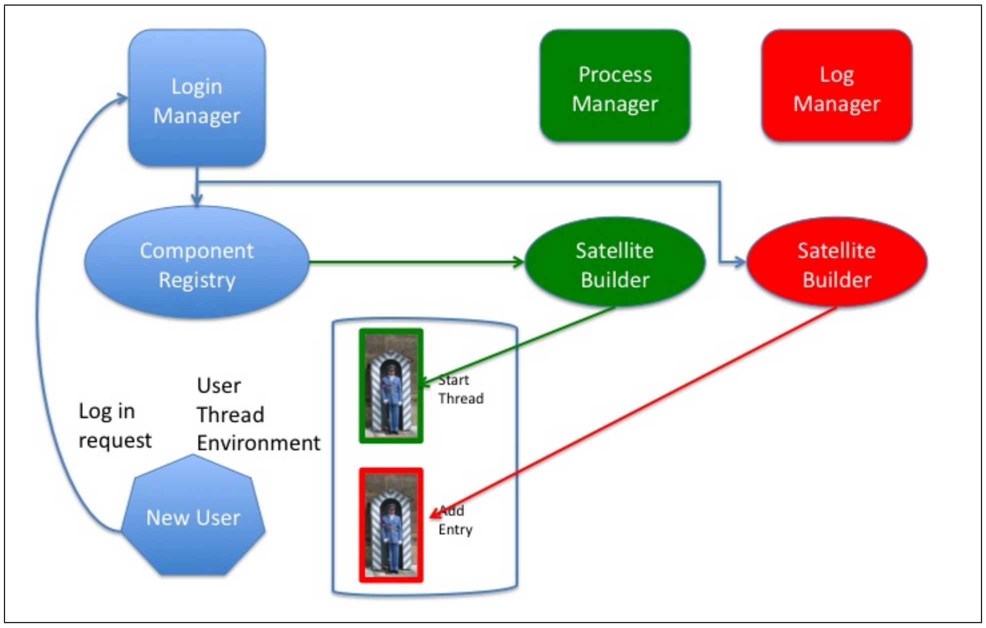


Figure 12: Starting A User Process

4.3 The Process-Manager

In this section we consider the Process-manager, the first of three "difficult" core components of ZKOS (the process-manager, the garbage-collector and the I/O system). The Process-manager (often termed the scheduler in many operating systems) is responsible for initiating and terminating user processes. In addition, it manages the division of processor time among separate threads of control[6]. In a conventional OS, the Process-manager executes within the kernel with unlimited privilege. There are several motivations for this: First, the scheduler clearly needs to be protected from accidental or intentional abuse from user code. Since it controls the allocation of system resources to processes, subversion of the Process-manager is tantamount to complete subversion of the system. Second, the Process-manager typically is responsible for saving and restoring the architectural state of the processor during process switches and this often involves the use of privileged instructions that are available only in the kernel mode of the processor. Finally, information used by the scheduler is sometimes stored within the user's address space, requiring the scheduler to have access to user memory.

As with the other ZKOS components we will consider, the Process-manager is a "globally" oriented facility; it's job is primarily the management of processor resources and time as a whole, rather than the rendering of services to individual processes. It, therefore, raises concerns about whether it can accidentally or through subversion become the vehicle for unintended information flows.

As we did with the Log-manager (and will do with the GC and I/O facilities) the first step in the design of the ZKOS Process-manager is a decomposition of its task into distinct sub-tasks each of which requires limited privileges. The key tasks fall into three categories:

- Allocation of processor time
- Saving and Restoring processor state during process switches
- Initiation, suspension, resumption and termination of processes.

Performing the first of these tasks clearly does not require the Process-manager to have access to any user data at all. In fact, all that is required is for the Process-manager to manage a data-structure keeping track of the system resources used by each process. This includes processor time, which the Process-manager can track by itself as it handles timer interrupts, which cause a process to relinquish the processor. Keeping track of other resource usage involves communication with user ZKOS components such as the I/O manager; again this requires no access to user data and is mediated by the satellite architecture described earlier in this section.

The second task, saving and restoring processor state, also need not involve unlimited access to user information. In fact, since we are co-designing the STA hardware with the ZKOS system, we can provide special capabilities for storing and restoring processor state that are distinct from the normal load/store instructions.

The third group really is just a collection of system services invoked using satellite compartments, principals, and gates as we've described above.

This leads to a hub and satellite architecture for the Process-manager in which the core manages the private data structures of the Process-manager, including resource consumption tables and scheduling queues. Conceptually, the part of this core that computes priorities and the like can run in a separate thread; this is particularly useful if the ZKOS system is supported by a multi-processor STA system.

[6]Unlike many systems that make a distinction between processes and threads, we will use the terms interchangeably. In many operating systems a process has a separate address space and a collection of threads that execute within that address space. However, in ZKOS there is a single, flat address space, making the distinction irrelevant

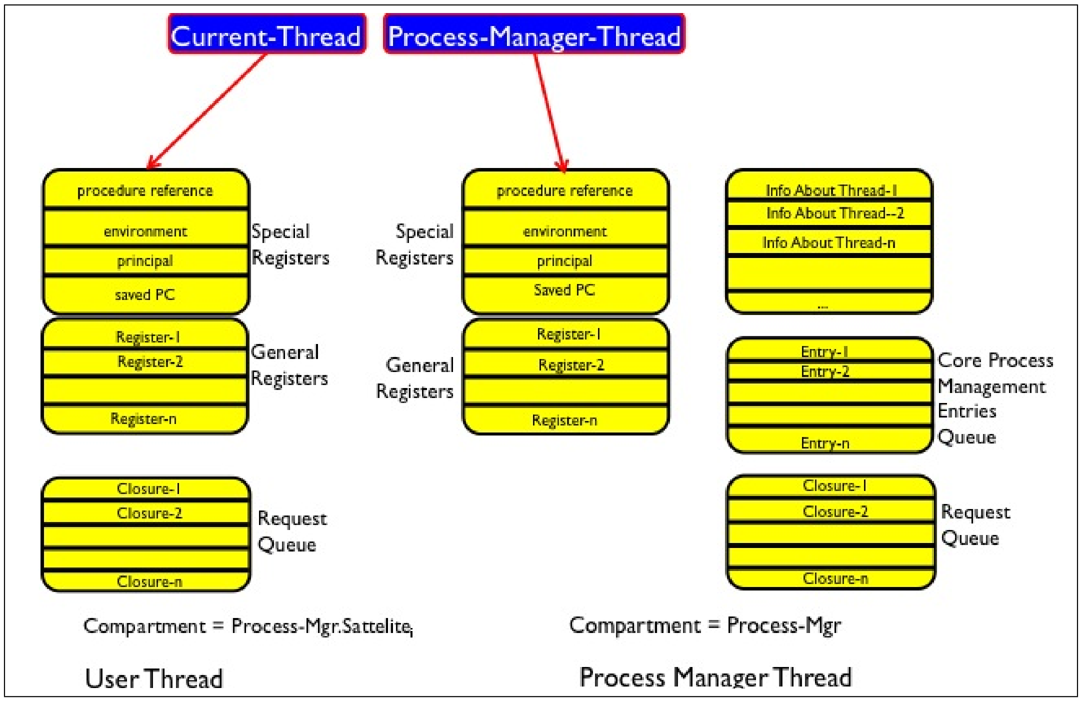


Figure 13: The Process-Manager's Data-Structures

Associated with each thread is a satellite compartment used to hold data particular to that thread. This includes a "processor-state" structure used to hold the suspended state of the thread, including the thread's current principal, compartment, procedure base, and the PC at which the thread should resume execution. In addition, this structure holds the other special registers (call-frame, locals-frame, return-frame etc. see Section 5.7) and general-purpose registers.

This structure has a unique data-type (Processor-state); associated with it are two special instructions: Store-processor-state and Restore-processor-state. ZKOS access rules restrict these instructions to principals associated with the Process-manager; in particular, the only sanctioned use of these instructions is by the satellite principal on Processor-state structures and only on those within the satellite compartment. There is one additional processor register, the Current-thread register, whose value is not stored in these structures.

The Restore-processor-state instruction takes a reference to a thread's Processor-state structure; it sets the Current-thread register to this value and then restores the other process registers from the referenced Processor-state structure. In effect, the process-state structure acts like an extended Call-frame (see Section 5.7) to which control is returned and from which the processor state is restored. After execution of the Restore-processor-state instruction the resumed thread is in control of the processor. The Save-processor-state instruction is the dual of the Restore-processor-state instruction; it saves the processor's registers in the Processor-state structure pointed at by the Current-thread register.

The Process-manager itself executes as a thread. In fact, it is to this thread that control passes when a timer interrupt is fielded. Thus, within the Process-manager compartment there is another such Processor-state structure, reserved for access by the Process-manager's core principal. Another processor register (Process-manager-thread) points to this structure; it is initialized during the bootstrap of the Process-manager.

Timer interrupts are affected by forcing a Swap-process-state instruction to be executed (which stores the state in the Processor-state structure of the current thread) and then transfers control to the Process-manager core by setting the processor's state to that stored in the Process-manager's Processor-state structure.

Normal load/store instructions are not allowed on Process-state data structures. This together with the fact that Principals associated the Process-manager do have access to data in the user's compartments means that there is no opportunity for inadvertently moving information in unintended ways through the Process-manager. Thus, by taking advantage of our ability to co-design the STA hardware with the ZKOS system we can provide strong guarantees on the permitted information flows.

In order to manage the allocation of processor resources, the Process-manager maintains a variety of tables summarizing the recent and cumulative resource consumption of each thread. These tables are held in the Process-manager's core compartment and are inaccessible to other principals. As mentioned earlier, the information necessary to keep these tables current comes from two sources: The first is the processor timer register, to which the Process-manager has access; the second is provided by calls from the I/O manager using gates (like those describe above) to mediate the inter-component calls.

ZKOS provides for inter-process communication through cross thread calls. These are managed nearly identically to normal calls (see Section 5.7); a call-frame holding the arguments of the call is constructed. But rather than making the call immediately, the call-frame and the other arguments to the Call instruction are entered in a queue held in the callee thread's satellite compartment. These queues are shown in Figure 13. The calling thread is enabled to make an entry in this queue through the use of a gate provided by the callee thread; this gate switches to the principal and compartment of the callee thread, queues the call and then returns. All threads are responsible for polling the request queue periodically.

Since the Process-manager itself has an associated thread, normal user code can make queued cross-thread calls to the Process-manager in order to invoke a variety of services. This includes requests to suspend execution until some condition occurs, requests to create a new thread, requests to terminate execution, etc. The process manager queues these requests as they are made. Whenever the Process-manager thread gains control it first updates its tables, then it empties the request queue; next it polls an internal queue of threads that have suspended execution until a specific time, moving those that have reached their target time into the queue of threads waiting for the processor. Finally, the Process-manager picks the next thread waiting for the processor and transfer control to it.

4.4 Least Privilege Garbage Collection

The second "difficult" area of system software that we investigate in this section is that of global "garbage collection".

Can we provide automatic management of dynamic memory without a single, privileged domain that can arbitrarily read and rewrite data? In particular, can we decompose memory management into multiple tasks, and minimize the privileges required by each task so that no task, even if subverted, provides a vector to compromise the integrity of the system? What hardware support would be necessary to dynamically enforce this identified least privilege operation?

To address these questions, and to illustrate the kind of service decomposition and least-privilege compartmentalization enabled by STA, we consider the design of a simple garbage collector. Garbage collected systems remove the error-prone burden of manual memory management from the programmer, in the process eliminating a class of dangling pointer bugs that can be exploited to breach a system. The garbage collector is a global operation that must touch and rewrite all of memory. As such, it is a service that could end up with global privileges and present a single-point of failure for a security system.

Nonetheless, with careful design, it appears possible to limit the privilege of the garbage collector so that it does not need privileges to arbitrarily change memory. Even if one of the decomposed processes associated with the garbage collector is arbitrarily subverted, the process does not have adequate privilege to compromise the system integrity (read data, change data or structures). The compromised garbage collector may cease to reclaim memory, leading to fail-stop behavior.

The resulting garbage collector design is very different from conventional operating system servers. This radically different design is enabled by the STA hardware. This example illustrates how STA changes the rules for system software construction, allowing radical fine-grained service compartmentalization without compromising performance.

4.4.1 Note on Design Approach

We could simply put the garbage collector in one domain, give that domain global read/write privileges, and argue or prove that it is correct and bug free. However, we have a long history of software bugs in critical routines that provide vectors for system subversion.

Instead, we are considering a defense in depth approach that provides not just one, but multiple, overlapping barriers to breach. Each barrier should be simple and demonstrable and should be adequate to maintain integrity. Nonetheless, even if the "impossible" happens and a single mechanism is breached, we want the system integrity to remain. Ultimately, we might be able to quantify the number of breaches (impossible things) that must happen for the system integrity to actually fail.

We start with:

- Isolate compartments—data structures for this service can only be touched by these service routines.
- These service routines can only run the code associated with this service, and no other routine can run this service's code.
- This service code cannot be overwritten (prevent code injection).
- Each compartment is small, amenable to easy proofs.

The garbage collector would be all-powerful to move data around; with verified functionality and no injection, it should not provide a liability to the system integrity. This would be more than

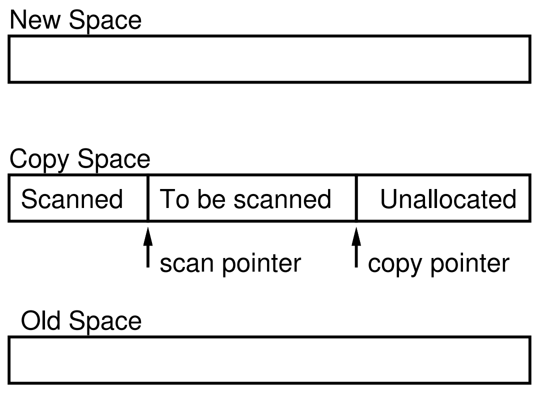


Figure 14: Space in Copying Garbage Collector

sufficient if no mechanism failed (no errors in our proofs, no mistakes in any hardware mechanisms, no policy errors,...). However, we want to further guarantee:

- This service can only manipulate application and OS data in a very specific way which does not change the meaning of the program. That is, even though we may move data around in memory, the data structure should never change and the security meaning of the metadata must be preserved.

This last point is broad, and we decompose it into specific requirements as we look at a particular garbage collection strategy in the following sections. The decomposition exploits a Clark-Wilson-style actor-verifier pattern [16] to help ensure robustness against compartment breaches.

4.4.2 Simple Garbage Collector Functionality

We consider a simple three-space copying garbage collector.

1. new-space – where new things are allocated
2. old-space – a region of memory the collector is reclaiming
3. copy-space – a place for data we are preserving (data being collected) to move as we evacuate old-space. This is further subdivided into three regions:
   (a) scanned – objects whose pointers are guaranteed to no longer point to old-space
   (b) to-be-scanned – objects that may still have pointers to old-space
   (c) unallocated – unused space before we copy live objects into it

By including forwarding pointers, this garbage collector can run as a separate thread interleaved with other programs. The semantics of a forwarding pointer are as follows: When data access encounters a forwarding pointer, the runtime system (some combination of hardware and system software) should perform another indirection, fetching the data at the destination of the forwarding pointer and providing that as the result instead. The forwarding pointer should be transparent to the application.

In the most straight-forward implementation, copy-space is a contiguous memory region. It is divided into unallocated, scanned, and to-be-scanned by two pointers:

1. copy-pointer

2. scan-pointer

Inside the copy-space, the three regions are ordered scanned, to-be-scanned, then unallocated. The two pointers define the boundaries between the regions. The copy-pointer points to the division between the to-be-scanned and unallocated spaces. The scan-pointer points to the division between the scanned and to-be-scanned.

4.4.2.1 Invariants

The garbage collector maintains the following invariants:

- Nothing in new-space can point to old-space
- Nothing in copy-space.scanned points to old-space
- Objects in copy-space.to-be-scanned may point to new-space, old-space, or copy-space

The garbage collector starts by initializing copy-space.to-be-scanned with the root set. When copy-space.to-be-scanned is empty, there are no references left to old-space and old-space can be reclaimed.

4.4.2.2 Algorithm

1. Start with copy space as all unallocated (*i.e.*, copy- and scan-pointers both point to the beginning of copy-space indicating both regions are empty).
2. Copy the "root pointers" to copy-space.to-be-scanned (*i.e.*, copy them to the copy-space.unallocated at the copy-pointer and advance the copy-pointer so that they are in copy-space.to-be-scanned)
3. While (copy-space.to-be-scanned is not empty) [equivalently, while the copy-pointer does not equal the scan pointer]
   (a) Scavenge: examine the first word in copy-space.to-be-scanned (*i.e.,* at the scan pointer)
      - If the word is a pointer to old-space,
        i. If the old-space target is a forwarding pointer, update the pointer to the destination of the forwarding pointer
        ii. If the old-space target is not a forwarding pointer,
          A. Transport the object (see subroutine below) pointed to by the pointer from old-space to the beginning of copy-space.to-be-scanned.
          B. Update the pointer (first word in copy-space.to-be-scanned) to the new location in copy-space
   (b) Re-assign this first word from copy-space.to-be-scanned to copy-space.scanned (*i.e.*, advance the scan pointer).
4. Condemn copy space. *i.e.* There are now no pointers to old-space. The space can be reclaimed. It is available to be reallocated as a future copy-space or new-space.

The Transport subroutine is as follows:
Given:

- pointer to an object in old-space. This is either the pointer which the copy-pointer points to, or a pointer referenced by a running application.
- pointer to a destination in copy-space.unallocated (*i.e.*, copy-pointer)

Operation (should appear atomic):

1. Copy object from old-space to copy-space.unallocated
2. Put the newly copied object in copy-space.to-be-scanned (*i.e.*, advance the copy-pointer)
3. Rewrite the old-space object with forwarding pointers to the relocated object in copy-space.

For applications to interleave, the runtime system must do two things:

1. follow forwarding pointers
2. when an application attempts to read a pointer to old-space, this should be replaced with a copy-space pointer, invoking the transporter as necessary so the object will have a valid copy-space address.

In conventional machines, forwarding pointers and attempts to load a pointer into old-space into a register generate traps to satisfy this requirement.

4.4.3 Garbage Collector Decomposition

We divide the garbage collector into 3 separate threads, each running in an independent and isolated domain.

1. Scavenger – performs the scavenging operation, including updating the copy-space.to-be-scanned pointer as it is converted to copy-space.scanned
2. Transporter – perform the transport operation
3. Condemner – condemns contents of old-space

We may also require:

- Allocator – to provide unallocated regions of memory (*e.g.*, to become copy-space, new-space) and to which we can return condemned space.
- GC Coordinator – to coordinate the swapping of regions between copy passes (i.e., the outer loop).
- Verifier – verify that copy-space.scanned does, in fact, contain no pointers to old-space
- Ager – mark a region as old-space

4.4.4 STA Hardware Support

We leverage the following capabilities provided by STA:

- Can tag a word in memory so it is only accessible by a particular domain.
- Can tag an instruction so that can only be executed by a particular domain.
- Can give a domain limited access (*e.g.*, read-only, write-only) to data in particular domains.
- Can limit instructions which a particular principal can execute and limit the instructions a particular principal can execute on particular domains.
- Can limit the the data permitted to be written into particular domains based on the metadata tags. (*e.g.*, here we will allow one principal to only write data tagged as forwarding pointers over data with a particular metadata tag).

4.4.5 STA Decomposed Garbage Collector Implementation

Each word in memory passes through a series of states as shown in Figure 15. Changes between these states are mediated by the vaious agents into which we decompose the task of the Garbage

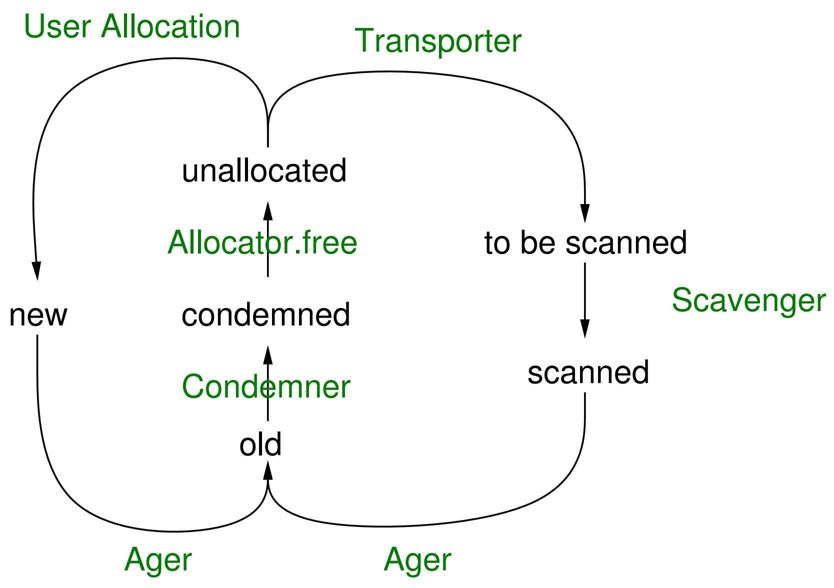


Figure 15: Memory Word Life Cycle and Agents that Change

Collector. These states help define limits on what each agent can do and can serve to detect misbehavior of any of the agents.

Scavenger
Needs:

- read the word pointed to by the scan-pointer (in copy-space.to-be-scanned)
- increment the scan-pointer
- overwrite the word pointed to by the scan-pointer
  - *n.b.*, this is the only word in copy-space.to-be-scanned that the scavenger needs to be able to write.
  - the overwritten word must have the same metadata (type, security compartment) as the word being overwritten.
  - when the word is a pointer, overwritten pointer must point to copy-space.to-be-scanned (or more specifically, the last object before the copy-pointer—however, if we allow application-driven transports to interleave with this process, then this may not be the last object allocated, so this may not be a viable invariant to enforce).
  - when the word is a non-pointer, the data should be unchanged.
  - possibly change the domain of this overwritten word from copy-space.to-be-scanned to copy-space.scanned

Algorithm:

1. read new-scan-pointer; block/wait if there is no new-scan-pointer to be read[7]

[7]Can use future/full-empty bit tagging on the new-scan-pointer input slot; the read will reset the new-scan-pointer slot to empty.

2. scan-pointer = new-scan-pointer
3. dereference scan-pointer
4. while (dereferenced scan-pointer is not an unallocated word)
   (a) if the dereferenced word is not a pointer, rewrite the word updating metadata to mark move from copy-space.to-be-scanned to copy-space.scanned
   (b) if the dereferenced word is a pointer
      i. call transporter on word
      ii. rewrite pointer changing pointer address to the new copy-space address returned by the transporter and updating metadata to mark move from copy-space.to-be-scanned to copy-space.scanned
   (c) increment scan-pointer
   (d) dereference scan-pointer
5. Send address of scan-pointer to GC Coordinator; this signals completion of this GC pass.

Ensuring least privilege:

- No access to old-space, copy-space.scanned, copy-space.unallocated, new-space
- Tag the new-scan-pointer as write-only by GC Coordinator, read-only by Scavenger.
- Tag the scan-pointer as private to the scavenger.
- Give the scan-pointer a designated domain/tag, and insert rules to limit use.
  1. allow increment
  2. The scan-pointer can only be overwritten (set) by a scan pointer.
  3. Only the GC Coordinator can make a pointer into a scan pointer.
  4. no other changes allowed
- Restrict the scavenger to only have read-write access to to copy-space.to-be-scanned for the word pointed to by the scan pointer. That is Read/write to copy-space.to-be-scanned is only permitted if the pointer through which the write is performed is tagged as a scan pointer.
- Provide rules to check the write to make sure the written word is suitably compatible with the word it is overwriting (only metadata change is copy-space.to-be-scanned to copy-space.scanned, and only data changed is pointer address).
- Limit state scavenger process can use.
- Disallow the scavenger to use any instruction which creates a pointer. (Presumably, this is the default for almost all processes.)

Subversion?: The scavenger has very limited ability to read or write anything. If it is locked down to not write anywhere else (which it has no need), it cannot steal the data it is copying.

- Could not advance the scan-pointer – This just stops reclaiming garbage which, worse-case, leads to a fail-stop condition.
- Could increment past pointers and not invoke transport – This is why we need a verifier routine to separately validate that copy-space.scanned has no pointers to old-space. Verifier will also depend on update of data to copy-space.to-be-scanned.
- Could insert the wrong pointer – It cannot create pointers, so is limited to pointers that it sees. Further, the limited state means it can only save a limited number of pointers around. If it uses an old-space pointer, then the Verifier will eventually catch it.

- Scavenger should not be given a new scan pointer until the old one is known to no longer be usable; that way, even if subverted and it tries to horde old scan pointers, it can never have two valid ones.

Transporter
Needs:

1. read old-space (potentially only as pointed to by old-space object pointer given)
2. write forwarding pointers into old-space (potentially only as pointed to by old-space object pointer given)
3. write into copy-space.unallocated (potentially only by copy to copy-pointer address)
4. advanced copy-pointer (potentially only by size of object being copied)

Algorithm: Input argument: old-object pointer (to old space object)

1. old-object = input-object-pointer
2. new-object-pointer = copy-pointer
3. copy-pointer = new-object-pointer + object length
4. perform a copy-operation from old-space object to new-object-pointer (copy operation actually retags old-space to copy-space.to-be-scanned)
5. write forwarding-pointer over old-space object
6. return new-object-pointer
7. if (new-copy-pointer is not empty)
   - copy-pointer = new-copy-pointer (and this marks new-copy-pointer at empty)

Last piece provides opportunity to update/reset copy-pointer. By making it part of the routine, we make sure the update doesn't interleave in a bad way. However, there could be a performance penalty for needing to always do this check. So, maybe a separate method which is restricted to not interleave with this one would provide better performance. Strictly speaking, it's only step 2 that needs to be locked out. However, these locking issues are known to be highly error prone, so the self-consuming version with fine-grained synchronization built into the communicated word seems easier to guarantee correctness.

Ensuring least privilege:

- new-copy-pointer is read-only for the transporter, write-only for the GC Coordinator.
- no access to copy-space.scanned, copy-space.to-be-scanned.
- copy-pointer is private to transporter.
- copy-pointer has own, designated tag type like scan-pointer with restriction rules on how it can be updated.
  1. allow update with an add (to advance over object)
  2. The copy-pointer can only be overwritten (set) by a copy pointer.
  3. Only the GC Coordinator can make a pointer into a copy pointer.
  4. no other changes allowed
- rules restricting transporter's writes to old-space to only be forwarding-pointers

- only operation to read old-space and write copy-space.unallocated is the copy-and-update-space instruction.[8]
- Limit state the transporter can have.
- Only the transporter is allowed to overwrite copy-space.unallocated.
- Transporter is only allowed to overwrite copy-space.unallocated.
- Data immediately following copy-space not tagged in same way. So, if the copy-space turns out to be too small, nothing is destructively overwritten.
- Disallow the transporter to use any instruction which creates a pointer. (Presumably, this is the default for almost all processes.)

Subversion?: The transporter also has very limited ability to read or write anything. If it is locked down to not write anything else (which it has no need), it cannot steal the data it is transporting.

- Could do nothing – If it doesn't return, we have a fail-stop condition. If it acts as the identify function, returning the old-space pointer it was given, even if the scavenger doesn't notice, the verifier will catch it.
- If it advanced the copy-pointer without copying, then the data there remains tagged as unallocated. When those pointers are dereferenced, the programs halts (fail-stop).
- If it advanced the copy-pointer too much, there will be unallocated words between objects. Using only type tags to delineate regions, the scavenger will erroneously think it has completed its task. This may require another checker (unallocated-verification) to make sure the scavenger wasn't tricked in this manner.
- If it advanced the copy-pointer too little, on the next invocation it will attempt to write over a word which is not in the copy-space.to-be-scanned region and that will flag as a violation.
- The transporter could write the wrong pointers for forwarding or return the wrong pointer – As with the scavenger, it cannot create pointers. It can only write pointers to which it already has access. The pointer it is writing is from the copy-pointer. So, if it saves away old copy-pointers, it can write/return those. Again, we have limited state which the transporter can retain.
  We could break this into two pieces. A stateless transport2 which takes two pointers, the old-object pointer and the new-space destination, and a transport1 routine which actually has access to the copy-pointer. Put these in different domains. Only transport2 can write forwarding pointers, and he's only ever given one at a time. Only transport1 gets to access the copy-pointer directly. If transport1 feeds transport2 an old copy-pointer, transport2 will fail on the copy operation. This gives the copy and forward pointer overwrite operations something like atomicity in that the routine is forced to use the same (consistent) addresses for copy and forwarding pointer writes.
- Transporter should not be given a new copy pointer until the old one is known to no longer be usable; that way, even if subverted and it tries to horde old copy pointers, it can never have two valid ones.

[8]Multiword copy instructions are also considered bad CISC instructions; at least this never has to worry about potential overlap which made this expensive on IBM 360 class machines. Think of it as a DMA. Another danger is VM page fault in the middle of the operation...needs to look atomic; we may not have VM pages, so this may be avoidable.

Condemner
Needs:

- write the condemned word over words tagged as in old-space

Algorithm:

1. read new-old-space-pointer; block/wait if there is no new-scan-pointer to be read
2. old-space = new-old-space-pointer
   (a) for each word in old-space
      i. write condemned word

Ensuring least privilege:

- new-old-space-pointer can be written only by GC Coordinator and read only by Condemner
- no access to new-space, copy-space
- no read access to old-space
- rules to allow to write over any old-space tagged word with the condemned tagged word

Subversion?

- could not condemn a word – when returned to allocator, allocator could catch. Allocator may need to require all words it gets are condemned.

GC Coordinator
Needs:

- ability to create new scan and copy pointers from existing pointers
- queue update to copy- and scan-pointers for transporter and scavenger
- queue old-space pointer for condemner
- mark region as old-space
- access to root set (or access to call another routine which has access to root set and have it dump them to copy-space)

Algorithm:

- repeat forever
  1. wait for scavenger to complete and return its end-of-scan scan-pointer
  2. invoke verifier on copy-space
  3. invoke unallocated-verification[9] on returned scan-pointer
  4. send allocator the scan-pointer (return unused suffix space)
  5. invoke condemner on old-space
  6. send allocator old-space pointer (to allow it to reclaim)
  7. old-space = pick a region to gc
  8. copy-space = call allocator to get region of size old-space.size
  9. make copy-pointer from copy-space pointer
  10. queue copy-pointer to transporter

[9]This is a routine to check that the scavenger wasn't tricked into returning prematurely. See Section 4.4.5

11. invoke ager on newly declared old-space region (marks it as old-space)
12. invoke transporter on root-set
13. make scan-pointer from copy-space pointer
14. queue scan-pointer to scavenger

Ensuring least privilege:

- all calls are linked procedure calls exclusively between these principals.
- Only GC Coordinator can create scan/copy pointers
- The GC should only give the scavenger and transporter new scan and copy pointers after it verifies the old ones are no longer useful; otherwise they could horde scan/copy pointers. This is why the code above is ordered as it is.

Subversion?:

- The choice of region to GC is a performance issue. It should be able to pick just about any region currently tagged as new-space or copy-space.scanned.
- Transporter and scavenger both have very specific requirements on how their space is tagged. Giving them wrong pointers should result in failure.
- The GC Coordinator could call condemner prematurely, without allowing scavenger to complete (or even calling the scavenger) and/or without getting a validation from the verifier. The result would be pointers to condemned space, which would be a fail-stop condition. If it got through condemning and return to allocation, the space could become unallocated. That would be OK, too, as access to an unallocated word would be a fail-stop condition.

In general, this seems like a powerful operation. However, the least-privilege limits in the things it invokes appear to constrain very tightly what it can do.

4.4.6 Discussion

- So far, there appear to be pulls for some very CISC-like instructions. Certainly, the old timers have complained the RISC instructions which offer fewer atomic operations made security harder, so maybe this is to be expected. ...definitely something to ponder...
- Write-once pointer?
- Split transporter pattern is intriguing and may merit further thought/development.

4.4.7 Variants

4.4.7.1 Indirect Region Tags Marking a region as old-space could require that we touch every word in that space. An alternate solution is to give regions tags whose interpretation changes. So, a new-space or copy-space.scanned tag could be reinterpreted as an old-space tag by simply changing the rules.

4.4.7.2 Address Regions The algorithm demands that we have limited access into various spaces. The description tries to be somewhat neutral as to how the spaces are delineated. Tags are one way to define the regions. Using registers with address bounds is another—this might require some dedicated microarchitectural registers that hold these bounds; hardware then could check bounds on accesses. Page permission might be another (though doesn't provide fine-grained separation between regions in copy-space).

4.5 Input and Output from Processor

Metadata tags protect the flow of information within the processor. However, we must also be concerned about what happens when the information leaves the processors. Furthermore, we must address the issue of how metadata tags get created without giving one principal unlimited authority to create or manipulate tags and thereby creating a potential single-point of failure in the security system. Additionally, we would like to abstract STA microarchitecture decisions, like the physical size and encoding of the hardware tags, from the system. To address these issues, we must consider what the boundary is for tagged information within the processor and how we can assure integrity of data when crossing this boundary. We see that it demands a translation and how STA features can support this translation.

4.5.1 Challenges and Requirements

4.5.1.1 Trust No One We cannot trust data outside of the processor. The integrity of the metadata demands that it only be changed in controlled ways that the hardware is enforcing. If we were to send data with tags over the network, there is no guarantee any other machine or person seeing the tags would respect them. Similarly, we cannot trust the metadata with anything coming from the network. To be secure, this also goes for persistent storage (file system on hard disk or flash) and main memory (Dynamic Random Access Memory or DRAM). With physical access, someone could put the hard disk in a different machine that did not respect our metadata tags. More invasively, someone could intercept data to and from a memory external to our processor and ignore or modify our tags.

Consequently, a full, system solution must use some other scheme for guaranteeing metadata integrity and confidentiality when exiting the processor. The obvious examples are cryptographic encoding for confidentiality and hashes for integrity. This suggests we must maintain an invariant that data leaving the processor be encoded, and we must be able to decode data properly as it enters the processor.

Once we've accepted that we must not trust external devices (*e.g.* network, hard disk, flash), we also realize we need not trust the device drivers that manage them. This is particularly important in designing a system with no, single, all-powerful entity. That is, a conventional system trusts the device driver to mediate access to the device. This driver has freedom to read all the data on (from) the device and modify it. A subverted device driver would allow it to bypass our security mechanisms. This is part of what forces device drivers to be part of the all powerful kernel. Even in cases where systems have moved device drivers out of the kernel, they still retain complete authority over their devices. This leads us to demand:

- Data flowing to shared device drivers must be tagged "unclassified, safe for public consumption"
- Data flowing from shared device drivers must be treated with no authority or tagging.

To make this work, we assume that our cryptography is strong enough that we are willing to consider the encrypted text "unclassified, safe for public consumption" and we are able to re-establish the integrity of a collection of bits returning to us using the cryptography.

4.5.1.2 Raw Bit Aggregators Raising this a level, we note that any subsystem (software or hardware) that handles raw bits[10] should not be trusted. Before handing any data off to these raw

[10]Raw Seething Bits (Section 2.1)

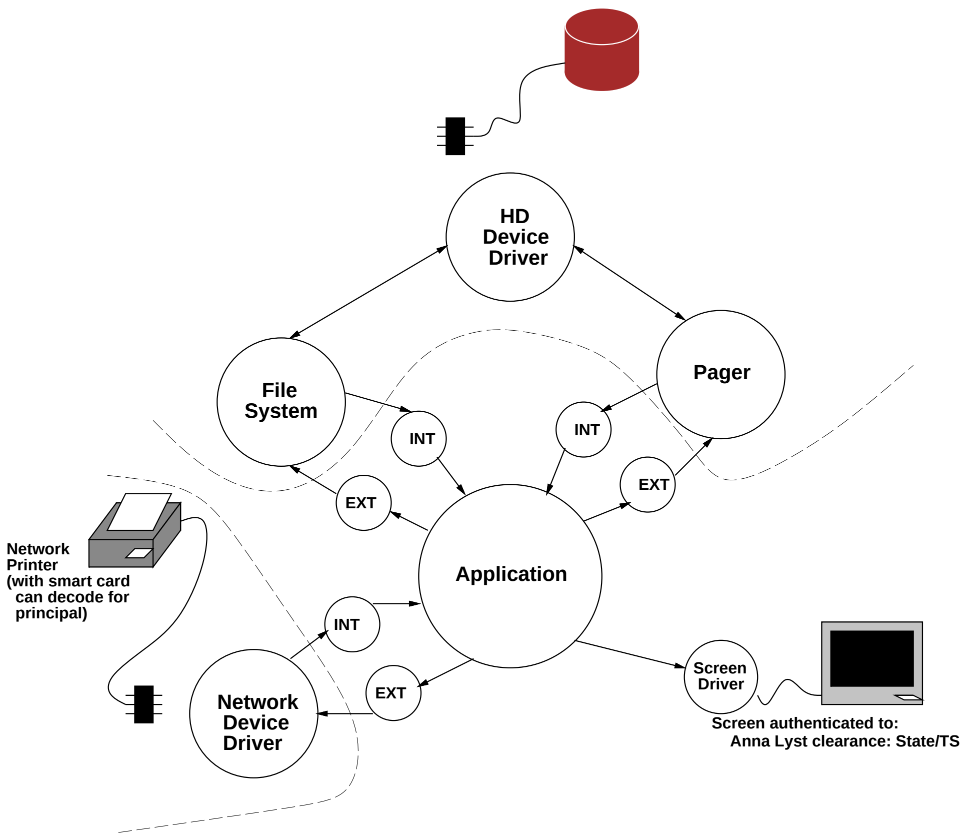

HD Device Driver
File System
Pager
INT
INT
EXT
EXT
Application
Network Printer (with smart card can decode for principal)
INT
EXT
Network Device Driver
Screen Driver
Screen authenticated to: Anna Lyst clearance: State/TS

Figure 16: Avoid Trusting Raw Bit Aggregators

bit aggregators, data should be encoded for external, untrusted handling. When receiving data from these aggregators, we must re-establish the trust and metadata. Figure 16 illustrates the intended conversions.

Using this discipline, it becomes possible to implement aggregate resource management services such as the memory pager, network stack, and file system without demanding that these routines be trusted components. As described so far, these components can still lose data (not save it, not send it, not return it, overwrite it), but they cannot exfiltrate it or compromise its integrity. This is an important component of avoiding reliance on any single component in the design, and hence containing the impact of any possible breaches.

4.5.1.3 Abstracting STA Microarchitecture It is common in processor architectures to define a stable, visible architecture that is abstracted from the detailed implementation or micro-architecture (*e.g.* [2]). This allows software to remain compatible across a large family of implementations. This is particularly important as the implementations evolve to exploit capacity delivered by Moore's Law. This demands that details of the processor that are likely to change (*e.g.* size of memories, relative speed of memories, depth of pipeline) be hidden from the software.

Similarly, a scalable STA should abstract out things that could and should change between implementations. The size and encoding of the metadata is one thing that might change. Section 5.6.2 discusses why the internal runtime representation of tags can be smaller than the external representations. We would like these internal capacities to be changeable among implementations.

Implementations that support limited tasks could get away with fewer principals and compartments than implementations that dealt with large ensembles of interacting software and agency coalitions. Similarly, the details of the TMU implementation should be changeable without affecting software. Just as we optimize the memory subsystem by changing the number of cache levels and details of cache implementation, we should be able to optimize the TMU implementation behind-the-scenes as well.

Accepting that we must convert between an external representation and an internal representation as developed above, we have a path for supporting this abstraction. That is, the internal metadata tag representation does not need to be exposed to any application software. Only the firmware associated with the tag management for a particular implementation is affected—similar to the way only the TLB miss handler needs to be cognizant of the detailed implementation of the TLB. All representations external to the processor are in the cryptographically encoded external representation. As data crosses the boundary between the internal, trusted region and the external it is recoded appropriately. This gives us:

- The internal, runtime tag encodings are independent of the external representation.

4.5.2 Device Driver Model

Communications with device drivers occurs using the satellite pattern (Section 7.1.3). The satellite principal is distinct from both the device driver and the application. As we'll see, we actually use multiple principals for the satellite. A gate is created between the application and the satellite and between the satellite and the device driver. The satellite has its own memory. It has access to two shared buffers, one associated with the application→satellite gate and one associated with the satellite→device driver gate. A comparable satellite and gate pair in the other direction serves for input from the device driver to the application. These satellites are responsible for converting between tagged data and encrypted data and vice versa.

4.5.3 Output Path

Figure 17 shows the decomposition of the output satellite for preparing data from the application to go to the device driver. In the extreme case, each of the nodes in the graph could be its own principal and compartment with limited privileges.

- Data enters as normal tagged data with a {Muti-Level Security (MLS) label,compartment,type} tag on data.

- **Tag2data** moves the tag into the data. This creates a word to represent the tag encoding. The tag-as-data word will still be tagged. The tag placed on this new word is a provisional rather than final tag.

- **Validate** looks at the proposed tag-as-data word and the original tagged word. If it agrees that is the correct tag, it upgrades the provisional tag to a real tag. This validate process is only allowed to upgrade provisional tags of this nature to real tags. Together these two processes provide Saltzer-Schroeder style *separation of privileges* [59] and act as a Clark-Wilson style [16] proposer-certifier pair.

- Once validated, the original word and the tag-as-data word are joined by the **compose** unit.

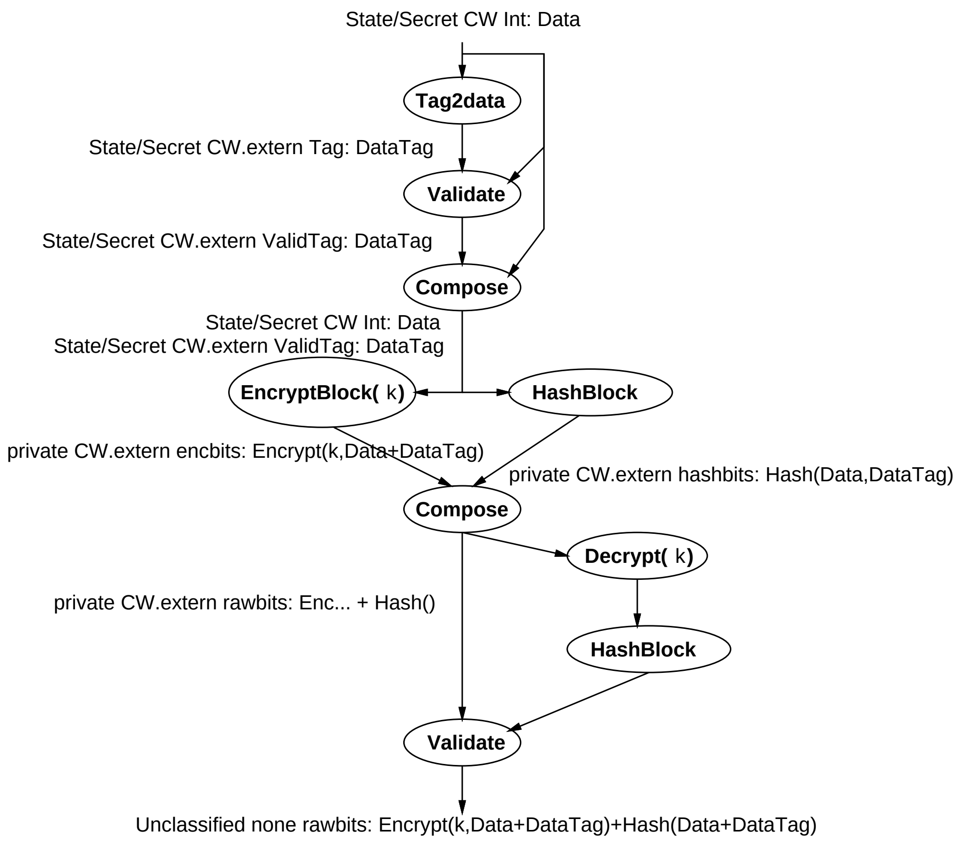

State/Secret CW Int: Data
Tag2data
State/Secret CW.extern Tag: DataTag
Validate
State/Secret CW.extern ValidTag: DataTag
Compose
State/Secret CW Int: Data
State/Secret CW.extern ValidTag: DataTag
EncryptBlock(k)
HashBlock
private CW.extern encbits: Encrypt(k,Data+DataTag)
private CW.extern hashbits: Hash(Data,DataTag)
Compose
Decrypt(k)
private CW.extern rawbits: Enc... + Hash()
HashBlock
Validate
Unclassified none rawbits: Encrypt(k,Data+DataTag)+Hash(Data+DataTag)

Figure 17: Data Flow for Output from Application to Device Driver

- At this point, we both encrypt the data in the **EncryptBlock** for confidentiality and compute a hash in the **HashBlock** for integrity and these bits are composed in a different **Compose** block. Data from the encryption is marked private to the satellite. At this point, the data is provisional and may not yet be safe for external visibility.

- A **Decrypt** and **HashBlock** are run to make sure that none of **EncryptBlock**, **HashBlock**, **Compose** tampered with the data. Note that the hash is an end-to-end check on the tag-as-data and data, guarding against the encryption task being unfaithful to the data it was given. If any one of these misbehaves, there will be a mismatch when decoded and checked.

- Finally, a **Validate** process checks that the decryption and hash decoding agree with the original block. If so, then the data can be retagged for external consumption. Once again, we've separated the privilege of proposing the final result and tagging, and the privilege to perform the critical retagging operation.

This output path makes use of special, privileged STA operations:

- tag2data—an operation to bring the tag representation into the datapath. This is more primitive than what the whole tag2data process performs. The process may further need to convert between the microarchitecture tag bits representing a smaller space of compartments and security levels on the processor and the richer, abstract representation for storage and communication.
- Retagging operations to validate and declassify information

The fine-grained privilege assignment supported by the TMU allows us to restrict which principals can perform the `tag2data` and retagging operations. Further, we can restrict the kind of tags they can operate upon. The retagging, for example, can only change one particular tag value into another.

4.5.4 Input Path

The input satellite needs to perform the complementary conversion as shown in Figure 17. This path must have the ability to place tags on words. This is a critical operation to control to prevent the forging or rewriting of metadata tags.

- Data enters tagged as raw bits with no privileges. We cannot trust the provenance or integrity of this data. If it is data produced by the processor, it will be suitably encrypted and hashed.

- We need to **Decrypt** and **HashBlock** the data.

- We **Validate** the hash on the decrypted data to make sure both that the data wasn't tampered with and that the decryption process is being faithful to the encrypted data.

- We then attempt to convert the tag-as-data back to a proper tag using **Data Tag**. Being able to write tags is a very powerful operation we must carefully control. If this process could write arbitrary tags, its subversion could be a single-point-of-failure. This routine is only allowed to create tags of a provisional nature. The rest of the system should not recognize or honor these provisional tags.

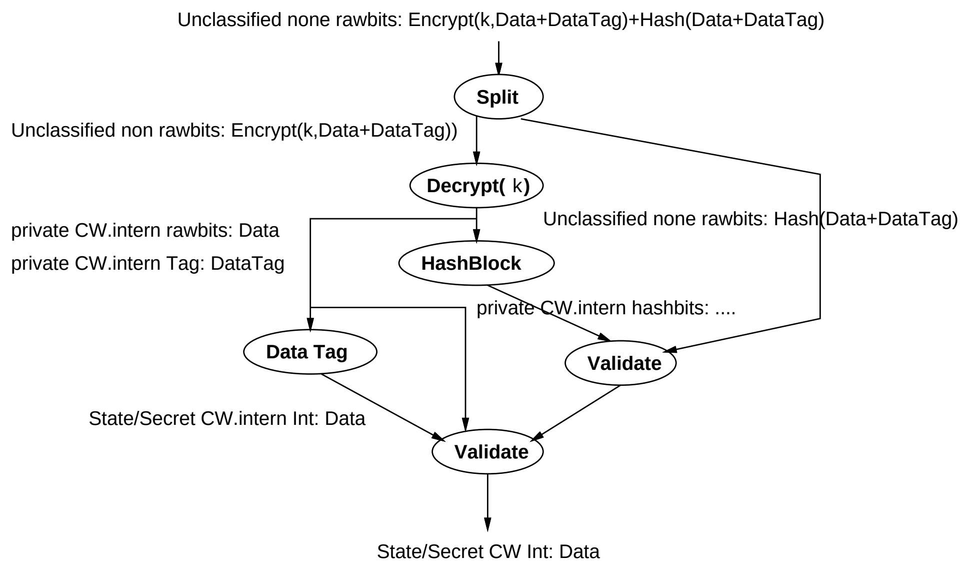

Unclassified none rawbits: Encrypt(k,Data+DataTag)+Hash(Data+DataTag)
Split
Unclassified non rawbits: Encrypt(k,Data+DataTag))
Decrypt(k)
Unclassified none rawbits: Hash(Data+DataTag)
private CW.intern rawbits: Data
private CW.intern Tag: DataTag
HashBlock
private CW.intern hashbits: ....
Data Tag
Validate
State/Secret CW.intern Int: Data
Validate
State/Secret CW Int: Data

Figure 18: Data Flow for Input from Device Driver to Application

- A final **Validate** routine now compares these provisional tagging of the word with the validated external tag representation. If they match, it can promote these provisional tags to real tags, again in the proposer-verifier style. The **validate** routine is only allowed to upgrade corresponding provisional to real tags. It does not have the ability to write arbitrary tags.

This input path makes use of special, privileged STA operations:

- data2tag—an operation to write a tag based on data; as noted this is a powerful operation. We use the rule set to limit the process to creating provisional tags, then the validation routine to check the tags created are faithful to the stored representation.
- tag2data—as introduced above, this performs the opposite operation as data2tag and is used by **Validate** to assure that **Data Tag**'s proposed tags are correct.
- Retagging operations to validate and declassify information

Again, fine-grained privilege assignment allows us to restrict both which principals can perform these operations and the types of tags they can operate upon and produce.

5 Security Tagged Architecture (STA) Hardware

5.1 Runtime Execution Model

We associate each word handled by the processor (register file, cache, memory) with a metadata tag. This tag carries semantic information about the word during runtime. In the simplest case this tag denotes the type of the data (*e.g.*, pointer, integer, float, instruction) and the compartment, which is used to identify those principals allowed to read or modify this word. We can create special types with restricted use contexts (*e.g.*, we can create a special new-space-pointer type for the garbage collector and restrict which principals can access this register and limit those principals to only be able to read or increment this pointer). The tag may also carry Mandatory Access Control security information (*e.g.*, unprivileged, secret, top-secret), integrity information, pedigree, and taint information (*e.g.*, this came unencrypted from the network). Except for a few privileged operations, the tag is neither readable nor writable explicitly but is calculated by the hardware based on the tags of the inputs.

When the processor executes an instruction, it computes the tag of the result word as well as checks that the operation being performed on the data is semantically meaningful and permissible. In this way, the tags on values are computed implicitly based on the inputs to the instruction and the semantics of the operation (*e.g.*, the sum of two floats is a float, the increment of a pointer is a pointer). The processor state includes special registers denoting the current principal and the bounds of the code block that it is running. The processor examines (1) the tags on the source, sink, and result operands, (2) the instruction, (3) the program counter, (4) the principal on whose behalf the code is executing, and (5) tags on the program counter, to determine if the operation is valid and to compute the result tag. Nonsensical operations (*e.g.*, adding two pointers, executing data, dereferencing through an instruction) are identified and cause a security trap; similarly operations disallowed by policy (*e.g.*, executing a privileged instruction to which the current principal does not have access, execution outside of the current code block, writing into a read-only compartment) also cause traps.

Principals are only changed at procedure call boundaries by invoking a “gate,” a combination of a procedure and a principal. During the execution of the procedure, the principal is bound to that of the gate and is restored to its previous value upon the procedure’s return. Gates, like all other objects, have a compartment, and access rules constrain the set of principals allowed to invoke a gate within a compartment. Thus the ability to change privileges is mediated by hardware checks. Furthermore, when a thread changes principal, this does not reflect an *increase* of privilege, but rather a transition to a *different* set of privileges appropriate to the task being carried out by the gate’s procedure. Each principal has only the *least privilege* necessary for its task. Access rules can also restrict all but a specific principal from accessing words with specific data types (*e.g.*, pointers used for storage management).

STA Microarchitecture: The simplest way to support the tagged words is to make the memory wider and associate a dedicated set of tag bits with every word. Alternate approaches include [23, 75, 77]. To support the model semantics above, we add a Tag Management Unit (TMU) in parallel with the normal execution path (See Figure 19). The operand tag information, principal, program counter, and instruction are all routed into this TMU. The TMU can be implemented as a Hash Execution Unit [14] (similar to a cache or TLB), hashing its inputs to calculate an address, and fetching an entry from an internal memory that includes a trap signal if the operation is illegal or the appropriate tag for the operation otherwise. The tag for the result word is joined with the result from the normal processor datapath before writeback to the register file or memory. Since the TMU executes in parallel with the processor datapath it does not slow processor execution and will typically have several processor pipeline stages for evaluation. Errors flagged by the TMU can squash writeback before the architectural state is modified, leveraging existing microarchitectural support for branch misprediction and exceptions.

The TMU implements the tag update and validation rules. The TMU is programmable to allow different rule sets to match the enforcement of different policies and software architectures. The TMU can be managed like a cache or TLB so that the TMU need only be as large as the current working set. While we might think of the TMU as being one monolithic table that takes in all of the aforementioned inputs and produces tag outputs and security exceptions, the implementation can be decomposed into a number of small tables each of which sees a smaller set of inputs. A rough estimate developed in our preliminary work suggest that the total size of these memories is about $2x10^5$ SRAM (Static Random Access Memory) bits, or less than the area of a 24KB L1 data cache. The L1 data caches on modern Opterons is 64KB and, itself, makes up only a tiny fraction of the area for the Opteron processor core. Note that the TMU tables are comparatively small (no larger than TLBs and register files in common use in today's processors) suggesting they will place no new constraint on processor cycle time.

Metadata tagging and computation allow the processor to collaborate with the OS to enforce meaningful and secure computations.

1. Type tagging on instructions prevent the system from being tricked into running data as code.
2. Principal-limited instructions allow us to avoid a single, all-powerful principal; the privileged operations can be divided and assigned in a fine-grained manner. Coupled with typing, the fine-grained privilege assignment may be further restricted (*e.g.*, a principal can be granted privilege to perform an operation only on certain, specially tagged data types).
3. Code block limitations and principal limited code ranges prevent a principal from being tricked into running other code with its privileges; similarly principal limited code prevents a different principal from running the code.
4. Code is tagged read-only, preventing overwrite. Multiple instruction type tags allow multiple levels of trust (*e.g.*, code written by an application can be sandboxed and treated distinctly from OS code validated at boot time.)
5. Bounds enforcement can eliminate buffer overflows.
6. Restricting the types upon which an instruction may operate can enforce abstractions (*e.g.*, normal instructions cannot overwrite a word tagged as a return address pointer)

The fine-grained hardware support afforded by STA makes a qualitative difference which transforms the cost landscape. STA:

- Reduces the cost of moving between isolation compartments from a context switch that may take 1000s of cycles to a procedure call that will take fewer than 10 cycles.
- Reduces the compartment size required to achieve high efficiency. With domain crossing around 10 cycles, it becomes reasonable to switch compartments to only run 10–100 instructions.
- Reduces the size of the dataset which a program or service can touch (accidental, through a bug, or through subversion) from an entire address space of megabytes or gigabytes down to a small compartment of kilobytes.

5.2 Ensuring Secure Information Flow

In TIARA, the machine word is the unit of information: each slot of memory and each register contains one word. A word consists of a tag and a value where the tag encodes the data-type and security context. Words are read and written atomically. The program counter associated with a process is tagged with a security class just like any other word. The "Principal Register" of the processor holds a word representing the privileges of the currently running process.

The security context part the tag is used to encode some aggregation of data that is to be treated uniformly from the point of view of security and information flow policy. Security contexts form a lattice; we denote by lub(A,B) the least upper bound in this lattice of the tags A and B.

An information flow policy is simply a set of rules such as A $\rightarrow$ B stating that information is allowed to flow from A to B; these rules are transitive. Flows not included in the transitive closure of the stated rules are disallowed. One goal of the TIARA hardware is to guarantee that these rules are followed. A more detailed treatment of this is provided in [14]; we provide an overview of the approach here.

As each instruction is executed, the TIARA hardware first checks that the operation is consistent with its policies; if so, it then computes a new security tag for both the program counter and the new result. The rules for the new tags are as follows:

- On a non-branching instruction I, acting on operands A and B:
  - The tag of the result is lub(tag(A), tag(B), tag(PC), tag(I)).
  - The tag of the PC is lub(tag(PC), tag(I))
- On a non-conditional branch instruction I: The new tag of the PC is lub(tag(PC), tag(I)).
- On a conditional branch instruction I, branching on data A: the tag of the PC is lub(tag(A), tag(PC), tag(I))

Whenever the processor takes a conditional branch, the tag of the current PC is pushed on an internal stack. When execution rejoins after the branch, this stack is popped and the tag of the PC is restored to its earlier value. This guarantees that the security context of the PC strictly represents the contexts of the precise set of data that has influenced the flow of control.

The basic information flow constraint is that no process can read a data value unless the flow policy allows information to flow from the security context of the datum being read to the security context of the the Principal.

Information flow policy also deal with I/O channels. Both input and output channels are assigned security context labels. Data coming in through an input channel is tagged with the channel's security context and may only be loaded into the processor if the information flow policy allows flows from the security context of the channel to the security context of the Principal of the current process. No data may be written to an output channel unless the information flow policy allows data to flow from the security context indicated by the datum's tag to that indicated by the I/O channel's tag.

A principal may be authorized to temporarily replace itself with another principal as the principal upon whose behalf the current process is running. We implement this with a hardware-supported "role" stack. Using this mechanism, we can implement Role-Based Access Control [26, 60], – a principal may take on any of a variety of "roles" (i.e. other principals), without ever having the access-rights of more than one of those roles at a time. The role stack is also quite useful for logging exactly which principals performed what actions in what roles.

5.3 The TMU

The key to implementing all of this is to quickly compute the new tags of the PC and the result while checking that the current operation is consistent with information flow and access control policies. We do this using a lookup table scheme that is supported by the TIARA hardware.

There have been several previous tagged computer systems that used tag bits to make the types of data manifest at runtime. Going far back, the Burroughs 5500 and its successors, used 3 tag bits (and a 48-bit address field) to encode a range of primitive data types; the Lisp machine [46, 24] used a full 8-bit tag. (The Intel-432 and the IBM series 38 machines also employ tagging systems).

However, TIARA uses a completely novel approach that is considerably more flexible and that deals with both data typing and security issues simultaneously. This is the Tag Managment Unit (TMU). The TMU is similar to a data cache or a TLB that operates on tags (i.e. on the type and security context of the data). Under program control, certain fields of the operands, the PC, and the instruction are identified as "tag bits" that encode data types, and security context. The TMU extracts these bit-fields and maps and uses them to access a cache of the hardware interlock rules in effect for the computation. The data retrieved from the cache contains a "trap flag" indicating whether the operation being attempted is prohibited; if so the operation is aborted and the processor is dispatched to an appropriate error handling routine. Otherwise, the information retrieved by the TMU includes the data type and security context of the result; these are combined with the result produced by the ALU and stored back into the register file. Similarly the updated tag of the PC is calculated and written back.

Like a cache, the TMU unit may miss; i.e. it might encounter a set of principal, tags and instructions that it hasn't yet seen. In this case it consults the *Policy Table* in main memory (just as a TLB would consult the page table) loads the entry into its memory and then resumes. As with other caches, the overhead of misses can be relatively large, but if the cache is of appropriate size, then misses will happen infrequently.

As an example, it is easy to see how multi-level security would be implemented within this framework: The tag of each operand includes a field that encodes the security level of the data, while the TMU lookup finds the maximum of the two input tags and inserts that into the result. Thus, each data word automatically takes on the highest security level of all the input data. On a load operation the TMU would consider the security tags of the object reference as well as the principal register; if the principal register contains a value lower than that of the object reference, then a trap signal is emitted, otherwise the operation proceeds.

The contents of the memories addressed by the TMU may be reloaded dynamically to support varying needs of the computation. In particular, instituting a change in security policy on the fly involves little more than flushing the HEX memories and swapping the pointer to the Policy Table. These mechanisms are describe in more detail in the technical reports presented in [14, 13].

Section 5.6 details implementation options for the TMU and estimates its size.

5.4 Support for Garbage Collection

TIARA's hardware must provide an object-oriented model of memory whose structuring conventions may not be violated. Rather than leaving storage management up the programmer, with the attendant risks of memory leaks and dangling pointers, TIARA instead provides for automatic, real-time garbage collection of its memory. The idea that garbage collection is a fundamental service is becoming widely adopted; for example, the Java Virtual Machine (JVM) and the Microsoft Common Language Runtime (CLR) are both garbage collected environments.

During our experience with the Lisp Machine project we developed several simple hardware features that allow the garbage collector to operate in real-time and that avoid any significant run-time overhead. The STA tag support effectively provides similar facilities; these involve little more than the ability to check each load or store operation for whether a word is a pointer data type (which is provided by the TMU) and if so whether the address points into a distinguished area of virtual memory. The TMU effects these hardware checks in parallel with the normal load/store datapath and suffice to erect the *read-barrier* and *write-barrier* of the GC algorithms that are documented fully in [44]. See Section 4.4 for details of the garbage collector implementation.

5.5 STA Processor Design

It is reasonable to think of the base STA Instruction Set Architecture (ISA) as a generic RISC ISA. As such, it has the standard arithmetic (*e.g.* add, sub, negate), logic (*e.g.* and, or, xor), load-store (*e.g.* load, load immediate, store, store immediate), branching (*e.g.* branch, branch zero) instructions and a conventional register set. STA adds a set of special processor registers to facilitate its secure operations. It also adds or redefines a set of special instructions for manipulating this state, managing memory and tags, and supporting procedure calls and gates.

5.5.1 STA Processor State

The added STA state supports the structured use of memory. Conventional architectures make managing procedure frames an issue for software and unify their state (*e.g.* base of frame register) with regular processor registers. To enforce the semantics of the procedural and object abstractions, we must make the semantics of these values and their operations visible to the architecture. Consequently, one important class of STA state represent the frame structure of memory, allowing STA along with a rule set to enforce the frame abstraction.

Conventional processors also have special mode registers in the processor for supporting privileged operations. Typically, this includes a binary user/kernel mode configuration. To provide separation of privileges and least privilege operation, STA instead has a more general notion of a principal, and the STA rules allow us to assign invidiual privileged operations (or even privileged operations on specific metadata types) to individual principals. This drives the inclusion of another set of state and instructions in STA.

Table 1 summarizes a set of registers we add to STA above the standard registers for intermediate computations. Frame registers contain both the base and the bound for the frame. This is used by the bounds checking for references relative to the registers to guarantee access stays inside the associated object. The procedure frame denotes the bounds of the code in the procedure, preventing execution or branches outside of the procedure. All branch operations are relative to the procedure base of frame (not the PC). All inter-procedure transfers demand call or return operations. There is no special system call instruction in STA. Rather, procedure calls can change principals upon entry (See Section 5.7.8) providing a more powerful, general mechanism that provides the isolation and principal change of system calls in the more general case without demanding an expensive context switch.

Section 5.7 details the use of this processor state.

5.5.2 STA Additional Instructions

Table 2 rounds up the special instructions in STA. These are described further in Section 6. Table 3 summarizes a set of allocation operations. These may or may not be primitive instructions in an STA implementation.

Table 1: STA Special Registers

Name	**Fields**	**Usage**
CALL_FRAME	base, bound	frame with arguments from caller
LOCALS_FRAME	base, bound	current operating frame—roughly equivalent to the current stack frame
RETURN_FRAME	base, bound	frame with return values from callee; this allows multiple value and unbounded object returns
OBJECT_FRAME	base, bound	frame for current object
PROCEDURE_FRAME	base, bound	frame for the code of the current procedure
ENVIRONMENT_FRAME	base, bound	frame for enclosing (typically lexical) environment for this procedure—potentially a chain of environments holding non-local value definitions
PRINCIPAL	principal	The entity on whose behave we are currently running. This helps determine the privileges assigned to operations performed.
COMPARTMENT	compartment	compartment into which new memory is allocated
PC	MLS, compartment, address	instruction pointer and its metadata tags
PREVIOUS_INSTRUCTION	instruction	the previous instruction to be used to guarantee that procedures can only be entered using the designated procedure call instruction and return entry points are only accessible from returns

Table 2: STA Special Instructions

Instruction	**Format**	**Operation**
CALL	2 reg	Call procedure at Rsrc with argument frame Rsrc2
RETURN	1 reg, 1 immed	RETURN_FRAME.BASE=src1; RETURN_FRAME.BOUNDS=src1+immed; Returns from called procedure
RETURN_ENTRY	none	finishes return
ENTRY	2 immed	LOCAL_FRAME.BASE=Newly allocated frame; LOCAL_FRAME.BOUNDS=LOCAL_FRAME.BASE+=immed1; PROCEDURE_BASE=PC; PROCEDURE_BASE=PC+immed2; check PREVIOUS_INSTRUCTION==CALL
STRMRD	2 reg	Rdst=stream(Rsrc1).next()
STRMWR	2 reg	stream(Rsrc).add(Rsrcc2)
STRMHD	2 reg	Rdst=(StreamHead)Rsrc1 (retagging)
STRMTL	2 reg	Rdst=(StreamTail)Rsrc1 (retagging)
TAG2DATA	2 reg	Rdst.data=Rsrc1.tag; Rdst.tag=Rsrc1.tag; mediated by TMU rules
DATA2TAG	3 reg	Rdst.data=Rsrc1; Rdst.tag=Rsrc2; mediated by TMU rules

Table 3: STA Psuedo-Instructions

Instruction	**Format**	**Operation**
ALLOCATE_GATE_FRAME	1 reg, 1 immed	Rdst=new gate frame size=immed
ALLOCATE_CALL_FRAME	1 reg, 1 immed	Rdst=new call frame size=immed
ALLOCATE_RETURN_FRAME	1 reg, 1 immed	Rdst=new return frame size=immed
ALLOCATE_ENVIRONMENT_FRAME	1 reg, 1 immed	Rdst=new environment frame size=immed
ALLOCATE_STRM	1 reg	Rdst=new stream

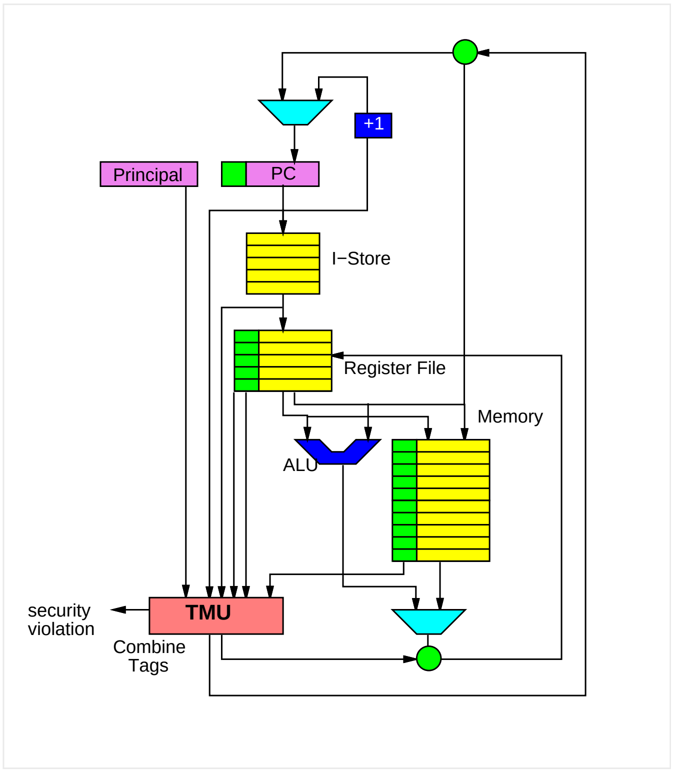


Figure 19: TMU Shown in Simple STA Datapath

5.6 TMU Estimates

Our initial hypothesis was that the metadata processing could be supported with modest hardware without impacting processor performance. To test this hypothesis, we examined the necessary composition for the Tag Management Unit (TMU) and estimate its size in several configurations. In this study, we find:

- The potential tag space is too large to manage with a monolithic, flat memory.
- Decomposing the TMU into components, the TMU needs around 192,600 SRAM-bit equivalents, or less than three-eighths the size of a modern 64K-Byte L1 Data cache.
- The TMU memories are comparable in size to modern register files or associative TLBs; this suggests the TMU will be no slower than these units that already fit within the processor's cycle time.

5.6.1 TMU Placement and I/O

The TMU processes the metadata tags in parallel with the execution datapath (See Figure 19). This means the TMU latency does not add to the processor cycle time. Its outputs are needed before the final writeback of register state to the register file. In a modern processor with support for speculation and out-of-order execution, the processor would already be prepared to squash instructions on mispredict before commiting them to architected state. Security errors in the TMU would be another input to this final commit or discard computation.

Figure 19 also shows the inputs and outputs that are potentially needed by the the TMU. These are summarized in Table 4.

Table 4: TMU I/O

	Field	**Number**
Input	Operand Data Tags	3
	Instruction Opcode	1
	MLS of Program Counter	1
	Value of Program Counter	1
	Principal	1
Output	Result Data Tag	1
	TMU Miss	1
	Security Violation	1

5.6.2 Tag and Field Sizes

To estimate the size of the TMU, we need to understand the size of the potential tag spaces. In doing so, it is useful to identify three different drivers of the size of the tag space:

1. Possible diversity—how many different cases do we need to represent? This drives the size of tag representations in the presistent store.
2. Operational epoch for one machine—how many different cases will one machine actually see? This is likely to be smaller than the full universe of cases all machines will see. This drives the size of the tag representation in the runtime.
3. Working set—how many different cases will one machine see over some period of time? This drives the desired size of the TMU caches. They should be large enough that misses are infrequent.

For the sake of elaboration, we assume the data tag is decomposed into three fields:

- Type—standard programming language data type (*e.g.* int, float, double, pointer); this can also represent some unique or differentiated types to aid in security and information flow (*e.g.* scan-pointer and forwarding-pointer in garbage collector (Section 4.4)).
- Compartment—ZKOS-level compartment
- MLS—Mandatory Access Control compartment (*e.g.* Secret, Top-Secret, TS:FBI)

Table 5 estimates the likely size of the tags and fields seen by the TMU. For ZKOS compartments, the estimate of 10 compartments per principal is consistent with the gate pattern (Section 5.7.8) where each application connects to 5–10 other applications or servers and has unique compartments associated with the interaction gates. We also assume the OS only needs 1000 address ranges that need special protection, and their code does need to be spread across the entire address space, so we can use fewer (40) bits to represent the important code regions that need to be distinguished for privileged operation.

5.6.3 Flat Rule Case

The first question we might ask is whether or not we can simply build a large, flat memory to hold the runtime rule set. Table 6 rounds up the number of entries in the table based on the

Table 5: STA Field Sizes

Field	**Possible**	**Runtime Rep.**	**Working Set**	**Bits**
Principal	Unbounded	1000	10	10
Opcodes	100	100	100	7
Critcal PC Values	2^{64}	1000	100	40
MLS	Unbounded	100	10	7
Compartment	per word	10/principal	10	14
Type	Unbounded	100	100	7
Miss	2	2	N/A	1
Security Violation	2	2	2	1

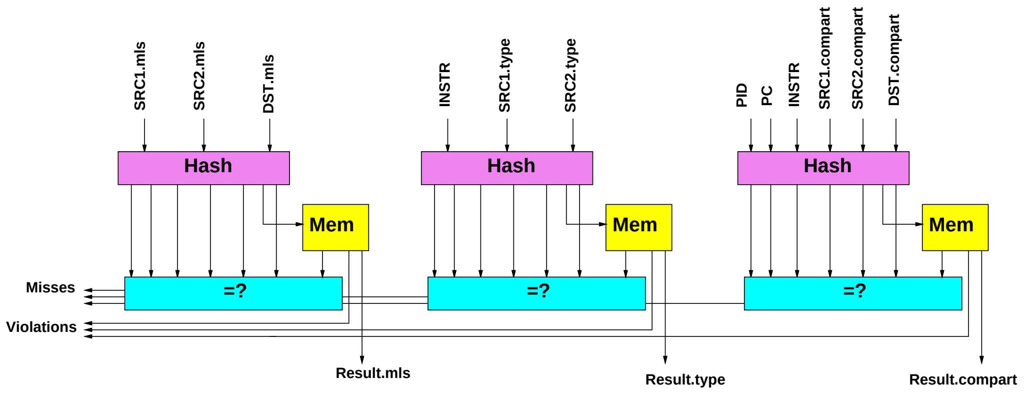


Figure 20: TMU Decomposition into Parallel Rule Tables

identified inputs and outputs above. Each of these entries would need to provide the 30 bits (MLS, Compartment, and Type for the result plus miss and security indication). Such a table would require about 2^{117} Bytes, which is not viable. This suggests it is important to exploit structure in the rules in order to engineer a viable TMU.

5.6.4 Rule Structure and Decompostion

The rules do not really require the full product space implied above. The tag fields (*e.g.* Type, MLS, Compartment) are mostly orthogonal. For example, read access rules only need to examine the principal and source compartment, and MLS lattice rules only need to look at MLS fields. This means we can reasonable decompose the function of the TMU into a number of separate units that operate in parallel. Table 7 summarizes the way fields interact.

From Table 7, we see that each of the 8 columns takes a subset of the inputs and produces a separate component of the TMU output. Figure 20 illustrates the general idea of how the TMU can be decomposed into these separate structures.

With this decomposition, Table 8 identifies how large each of these component tables need to be to satisfy the assumed working set from Table 5. As noted here, for typical, well-formed rules some cases are not multiplicative.

- Typically, the destination and source compartments will need to match.

Table 6: Entries Required for Flat Rules

Field	**Runtime Cases**
Principal	1000
PC Value	1000
PC.MLS	100
Instruction	100
Src1.MLS	100
Src1.compart	10^4
Src1.type	100
Src2.MLS	100
Src2.compart	10^4
Src2.type	100
Dst.MLS	100
Dst.compart	10^4
Dst.type	100
Total	10^{34}

Table 7: Sparse Structure of TMU Computation

Field	**Read**		**Store**		**Execute**	**MLS Lattice**	**Type**	**Principal Limited**
	Src1	Src2	MLS	Compart				
Principal	x	x		x	x			x
PC Value					x			
PC.MLS						x		
Instruction							x	x
Src1.MLS			x			x		
Src1.compart	x			x				
Src1.type							x	x
Src2.MLS						x		
Src2.compart		x						
Src2.type							x	x
Dst.MLS			x					
Dst.compart				x				
Dst.type								

Table 8: Size of Rule Sets in Decomposed Tables

Field	**Read**		**Store**		**Execute**	**MLS Lattice**	**Type**	**Principal Limited**
	Src1	Src2	MLS	Compart				
Principal	10	10		10	100			100
PC Value					†			
PC.MLS						10		
Instruction							100	†
Src1.MLS			10			10		
Src1.compart	10			10				
Src1.type							10	†
Src2.MLS						†		
Src2.compart		10						
Src2.type							†	†
Dst.MLS			10					
Dst.compart				†				
Dst.type								
Rules	100	100	100	100	100	100	1000	100

† – correlated with other cases; does not expand rule set size.

- Typically, each principal that has exclusive code will have one such code block.
- Typically, source and destination types will match, or be from a small set of choices.
- There will be a small set of total principal limited instruction cases. To make this restriction powerful, the rule can be based upon instruction and types, but generally each principal will just have one or a few cases allowed.

5.6.5 Decomposed PLA-Style TMU

The previous section shows that most of the tables have around 100 entries (smaller than modern TLBs), with the type table perhaps being as large as 1000 entries (ballpark of modern physical register files). To ground the size further, in this section, we consider building fully post-fabrication programmable logic arrays (PLAs) to support the logic and estimate their area.

The PLA for each decomposed rule table (*e.g.* Figure 21) takes as input, all the bits associated with the identified fields. Each product term (PTERM) in the PLA supports one rule and defines the ouptut in the case of that rule. For each input bit, the PTERM can be programmed to match-a-1, match-a-0, or as a "don't care" such that it matches either a zero or one input. In the simplest case, the PTERM performs an associative match. A triggered PTERM then selects its output case. A PLA with I inputs, O outputs, and P product terms requires:

$$A_{pla}(I,P,O) = (2I+O) \times P \times A_{xpoint} \quad (1)$$

A PLA crosspoint is roughly the area of two SRAM bits, A_{bit}. The inputs are multiplied by two because the PLA PTERMs need access to both polarities of each input signal.

Using the field sizes from Table 5, we can identify the number of input and output bits each decomposed TMU PLA must support. These are summarized in Table 9. Read, Store, and Execute

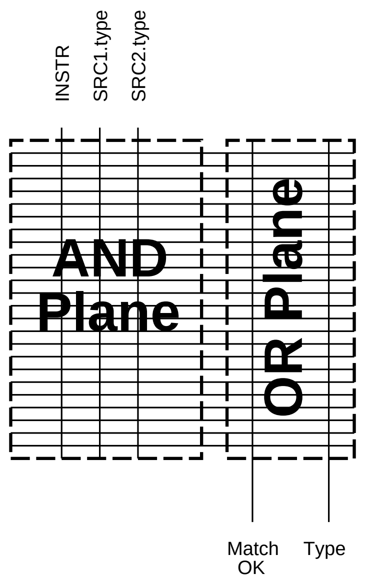


Figure 21: Composition of sample PLA Rule Table

rules only identify security violations or misses, so only output 2 bits. MLS and Type compute the resulting MLS or Type. Special, principal-limited instructions may define the entire tag, so they need the entire 28 bits of output for the tag as well as security violation and miss bits.

Table 10 estimates the area of each of the tables and summarizes the total area for the TMU. We see here that the TMU requires roughly the area of 192,600 SRAM bits, a little under 24KB. Note that this is 3/8th the area of the L1 D-cache on an AMD Opteron.

5.6.6 Future Optimization Options

The estimates above are for the most straightforward implementation. Options remain to further optimize rule encodings and TMU implementations, including:

- Special-case common idioms. For example, it is generally true that when input types are equal (input MLS), the result type (MLS) should match the common input value.
- Select encoding of fields to maximize sharing of PTERMS within the PLA.

In the simplest case, misses are refilled by refilling entries form main memory, like a TLB. If this makes miss-service time too large multiple levels of TMU could reduce common-case service times.

5.7 Procedure Call Architecture

5.7.1 Goals

This section describes the procedure call architecture of the TIARA processor, one of few areas in which the TIARA design differs strongly from conventional architectures. There are several goals motivating the design, including:

- Support for the thorough object-oriented, semantics enforcing approach of TIARA. As we will see, the TIARA design prevents a variety of semantics violations such as overwriting stack headers and branching into the middle of executable code.

Table 9: PLA I/O Audit

Field	Bits	**Read**		**Store**		**Execute**	**MLS Lattice**	**Type**	**Principal Limited**
		Src1	Src2	MLS	Compart				
Principal	10	x	x		x	x			x
PC Value	40					x			
PC.MLS	7						x		
Instruction	7							x	x
Src1.MLS	7			x			x		
Src1.compart	14	x			x				
Src1.type	7							x	x
Src2.MLS	7						x		
Src2.compart	14		x						
Src2.type	7							x	x
Dst.MLS	7			x					
Dst.compart	14				x				
Input Bits		24	24	14	38	50	21	21	31
Output Bits		2	2	2	2	2	9	9	30

Table 10: Decomposed TMU Area Estimate

Field	**Read**		**Store**		**Execute**	**MLS Lattice**	**Type**	**Principal Limited**	**Total**
	Src1	Src2	MLS	Compart					
Input Bits	24	24	14	38	50	21	21	31	
Output Bits	2	2	2	2	2	9	9	30	
Rules	100	100	100	100	100	100	1000	100	
Crosspoints	5000	5000	3000	7800	10200	5100	51000	9200	
SRAM Bits	10000	10000	6000	15600	20400	10200	102000	18400	192,600

- Support for the tag processing architecture of TIARA which involves systematic enforcement of access rights based on the identity of the executing principal as well as data types and compartments of the objects being operated on. This requires the processor to maintain within its architectural state a Principal Register that is maintained across procedure calls. Similarly, the processor maintains within its architectural state a register representing the compartment in which new objects are allocated.
- Support for modern programming language styles involving persistent "Closures" that cause stack frames to be retained after a procedure returns. Similarly, we have the goal of supporting multi-thread models in which procedure calls can spawn new threads or involve communication between threads.
- Support for isolation, compartmentalization, and access control between caller and callee.

5.7.2 The Problem

In conventional architectures, instructions are indistinguishable from other data. This facilitates many of the standard attack patterns used by attackers; these involve the following steps:

1. Place into memory a set of data whose bit patterns are identical to those of legitimate instructions. This can be done using a variety of techniques including:
    - "Buffer overflows", i.e. passing large byte strings into a routine that copies the string into fixed sized buffer. If the copy loop doesn't perform a bounds check, then it continues to copy the bytes into the memory adjacent to the buffer, overwriting whatever data had been present.
    - Allocating new storage legitimately (e.g. allocating an array) and then writing bit patterns into the new storage that is identical to the bit patterns of legitimate instructions. A common pattern is to fill the storage with "noop slides", strings of integers that when viewed as instructions encode noops followed by a branch to the payload target code.
2. Force control to jump into this block of data. Again there are standard techniques for doing this, including:
    - Stack smashing, in which a buffer overflow of a buffer allocated on the stack overwrites the return address in the stack header. When the procedure returns, instead of execution resuming in the calling procedure, control jumps into the attack payload code.
    - Dispatch abuse, in which a computed jump is passed invalid data that causes control to jump into the attack payload, rather than an intended dispatch target. One example of this is invoking a method on an invalid object (e.g. an integer), but there are a large number of variants on this theme

What is clear about this attack pattern is that it critically depends on violation of the intended semantics of the hardware and software systems, including:

- Violations of the semantics of the "basic object abstraction", in particular violations of the extent of objects to overflow buffers.

- Violations of the type abstraction of objects, in which integers are treated as instructions by the hardware.

- Violations of control abstractions, in which control is forced to jump to arbitrary places, including places outside of the currently executing procedure.

- Violations of intended access control rules in which stack headers which are intended to be immutable are, in fact, overwritten.

5.7.3 Code Objects and Procedure Pointers

The TIARA design is intended to prevent all of these violations, by using meta-data tags to systematically impose a highly object-oriented structure throughout the system. In TIARA, instructions are distinguishable from other data (because of their unique data-type tag); the processor will trap if the program counter points to data not tagged as instructions.

All TIARA instructions are contained within a specific type of object called a "code-object", see Figure 22. References to code-objects are also a specific type of data, called a "procedure reference". Code objects, like all objects, have a manifest extent and base. All branch instructions are relative to the base of the code-object and are bounds checked; attempting to branch out of the currently executing procedure is illegal and will cause a processor trap. Because the base and bounds of the current code-object are constantly being examined, the processor architectural state maintains a register (the Procedure Register) pointing to the currently executing code object.

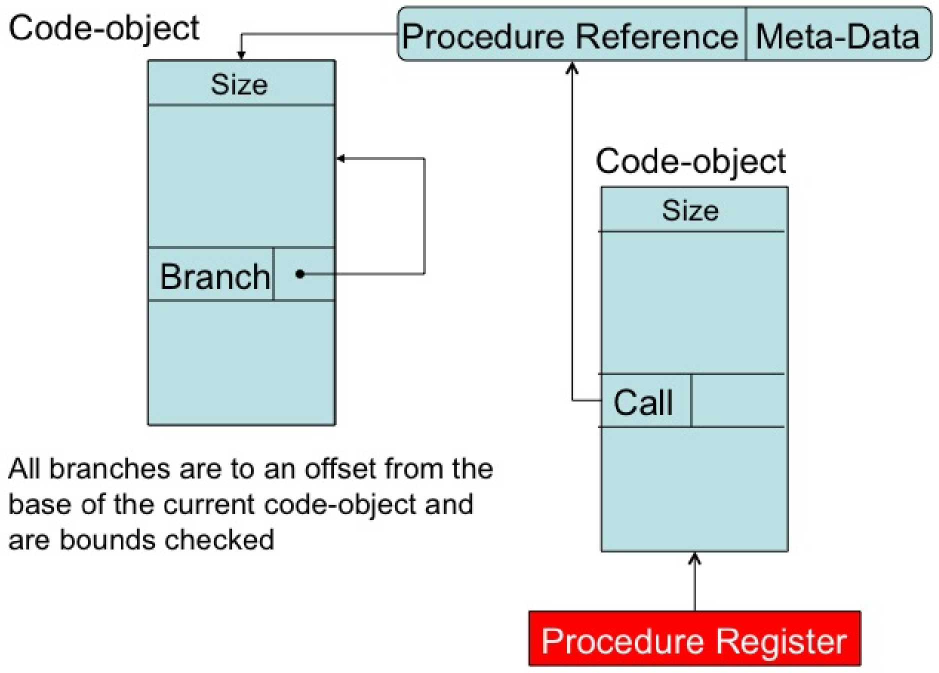

Code-object
Procedure Reference
Meta-Data
Size
Branch
Code-object
Size
Call
All branches are to an offset from the base of the current code-object and are bounds checked
Procedure Register

Figure 22: Code Objects

There are only three ways to effect a non-local transfer of control:

1. Procedure calls

2. Procedure return

3. Traps

Each of these is highly structured. Procedure call instructions can cause transfer of control only to the entry point of a code object. Returns can only cause transfer to the instruction following a call instruction. Traps are procedure calls forced by the hardware to trap handlers, which are procedures stored in the hardware trap vector.

Code objects have a standard structure: The first instruction in any code object is an Entry instruction. This is followed by a dispatch table, consisting of a set of jumps, one for each entry point of the procedure. Each entry point consists of a prelude set of instructions, peculiar to that entry point, followed by a branch to the instructions common to all entry points which follow the entry point preludes. Following the instructions are a set of constants used by the code-object; these can consist of raw data (e.g. integers, strings, symbols) but they can also consist of procedure references to other procedures. In particular, many languages have internal, lexically-scoped procedures that are referenced in the constants area of the code object. These structures are shown in Figure 23. Simple procedures have only a single entry point, which is the common case; multiple entry points can be used in a variety of ways including method dispatching, dispatching based on number of provided arguments.

5.7.4 Procedure Calls

As stated above, the procedure call instruction can transfer control only to the beginning of a procedure. As shown in Figure 23 the call instruction takes three arguments:

1. A procedure reference, that points to the code object being invoked. This must have data type, "procedure reference".

2. The entry point to invoke. This must be an integer

3. A reference to a "Call Frame". This has data type "call-frame-reference".

The process of making a call begins by allocating storage for a call frame that will hold the arguments that will be passed to the called procedure. Calls to the allocator are made using the Allocate instruction which takes a size and a data type as arguments, as well as the register number in which to return the reference to the newly created object. The allocate instruction creates the requested data structure, allocating it within the current compartment; it returns a reference to the newly allocated object. In the case of a call frame, the requested size is the number of arguments plus space for the architectural state that will be saved in the call frame. The caller then places the arguments into the call frame, loads a register with a reference to the code-block to be called, and then issues the call instruction.

The processor then saves the architectural state in the header section of the call frame, updates the registers constituting the architectural state to point to the new code-block and call-frame and then transfers control to the entry instruction of the new code-block. The entry instruction takes two arguments: The number of entry points, and the size of the "Locals Frame". In executing the entry instruction, the processor first checks that the entry point number passed in the call instruction is less than the number of entry points in the entry instruction, trapping if the entry number is too large. Second it allocates a data structure, called a Locals-Frame of the size specified in the entry instruction; a reference (of type locals-frame) to this data structure is placed in the

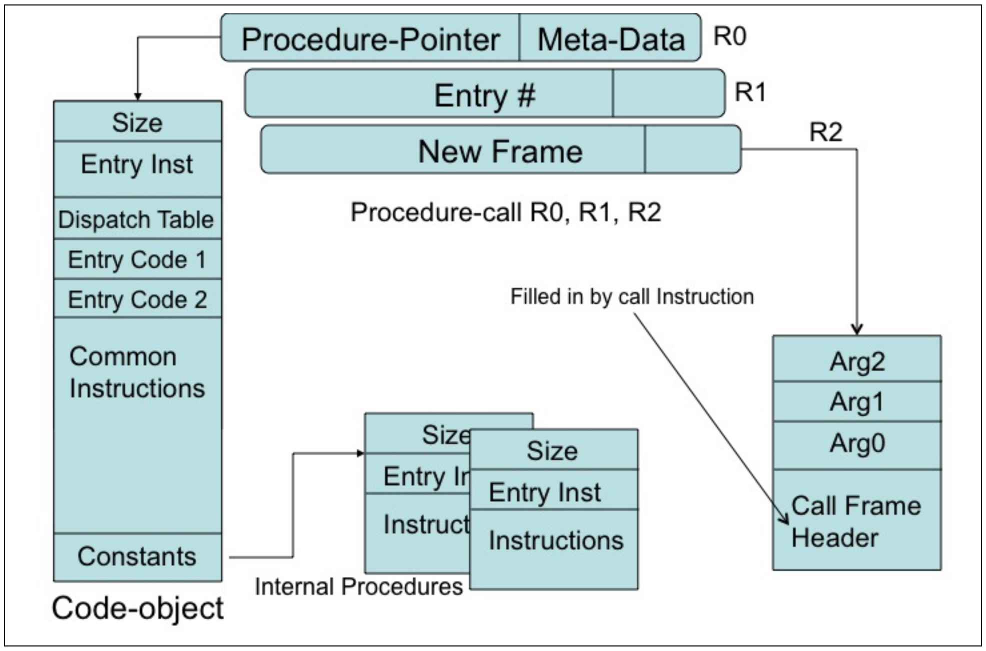


Figure 23: Making A Procedure Call

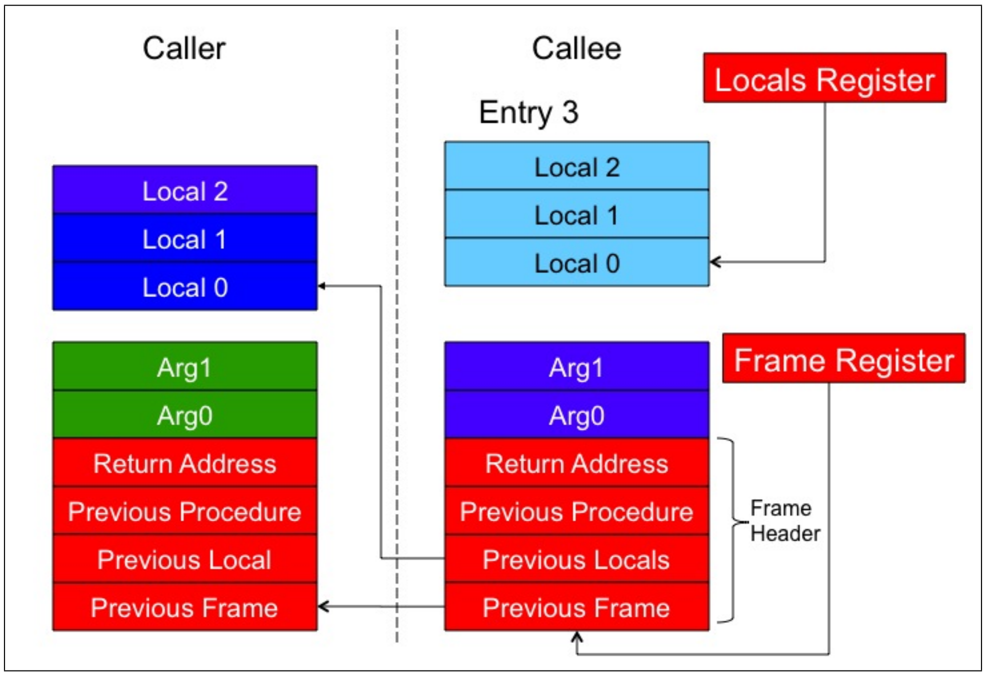


Figure 24: State After The Call

"Locals Register" of the processor. The locals-frame is allocated in the compartment that is current after the call; we will return to this subject later in Section 5.7.8. The state of the processor at this point is shown in Figure 24. (To avoid clutter we haven't shown the procedure-register and program-counter in this figure). The entry instruction then branches through the dispatch table to the appropriate entry point prelude call and the transfer of control is complete.

Several things that are different from conventional processors should be noted at this point:

- Instructions are distinguishable, via their unique data type tag from all other types of data; only data tagged as instructions are executable.
- Instructions only exist within code blocks, which are also distinguishable by their unique data type tag.
- Non-local transfer of control can only be affected via procedure call instructions and these can only transfer control to an entry point of a code-block
- Branch instructions are strictly local transfers within a code block. These are bounds checked and cannot transfer control to a location outside the code-block.
- There is no special stack, instead frames are allocated in the heap and are reclaimed only by garbage collection. "Closures" that are returned from a procedure invocation or that are stored in data structures will retain those stack frames involved in the closure; unlike conventional push-pop stack discipline, there is no need to evacuate a "retained frame" and no possibility of creating confusion about the identity of the storage allocated for the frame (i.e. no possibility of dangling pointers).
- The "stack" frame is "split". There are two parts: The call-frame part is allocated by the caller in its compartment and initialized with values that the caller is passing to the callee; The locals-frame is allocated by the callee in its compartment and is used by the callee to hold whatever temporary values it might need. We have not yet discussed how the callee's compartment might be different from that of the caller, we will turn to that subject in a later section. Nevertheless, it should be noted, that the caller has the ability to impose access restrictions on the data it passes to the callee because the compartment and principal of the caller might be different from that of the callee.
- The processor architectural state saved in the header of the call-frame consists of copies of processor registers, such as the program-counter, the frame-register and the locals-register. These all have unique data types (e.g. call-frame-reference, locals-frame-reference, return-address) and the standard access rules prohibit overwriting these types of data with any other type of data. Thus it is impossible to overwrite the return address stored in a stack frame.

5.7.5 Returning

Procedure return in TIARA is somewhat different from that used in conventional processors. In many ways, returning is similar to making a call. The first step is to allocate a "return-frame" in which one or more return values will be passed back to the caller of the currently executing procedure. The allocate instruction constructs an object of type return-frame, placing a reference to this object into a register specified as an argument to the allocate instruction. The return-frame

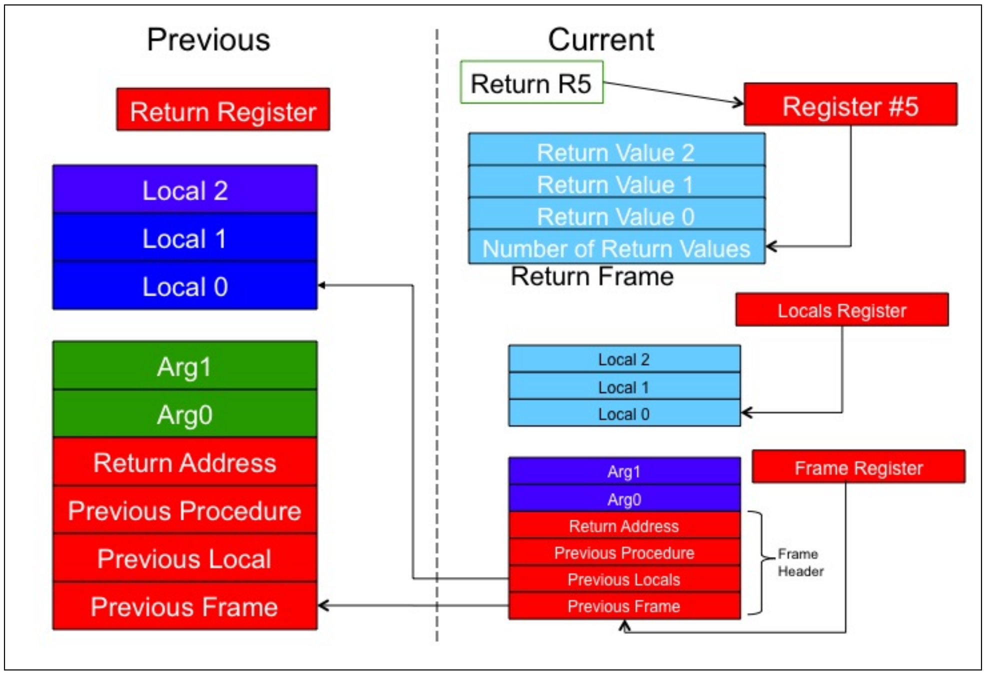


Figure 25: State Before Returning

is tagged as being in the callee's compartment. The currently executing procedure then places its return values into the return frame. The processor state at this time is shown in Figure 25.

The procedure then executes the return instruction which takes two arguments: The number of return values and the register referencing the return frame. The processor, at this point, restores the processor state from the currently executing procedures call-frame header, including the frame-register, locals-register, procedure-register and program counter. In addition, a previously unmentioned processor register, the RETURN_FRAME Register is updated to contain the reference to the return-frame included in the return instruction. The reason for this register is to remove the need for the caller and callee to agree on how to return values, just as the use of the frame-register removes the need for agreement on how to pass arguments. The state of the processor after returning is shown in Figure 26.

After the return instruction, the program counter points to the first instruction in the callee's code block after the call instruction. This is required to be a Return-Entry instruction. As with calls, the return instruction takes a third argument which is the return number; the return-entry instruction takes an argument which is number of return entry points. The return-entry instruction is followed by a dispatch table, preludes for each return and then common code that is branched to at the end of each prelude. One reason for such a dispatch is to allow separate return cases for normal and exceptional execution (*e.g.* catch-throw, try-throw). In effect, the process of returning is a call to the remainder of the calling procedure, passing arguments and entering at one of many possible entry points. However, the return-frame register, in contrast to the call-frame register is not maintained across calls as part of the processor architectural state. The procedure must instead copy the return-frame register into another register or explicitly save and restore it when necessary.

In summary, we have a highly object oriented framework in which instructions only exist in code objects, non-local transfers of control can only be made via procedure call and return, other branches are within the current code object and are bounds checked. Stack frames are allocated in

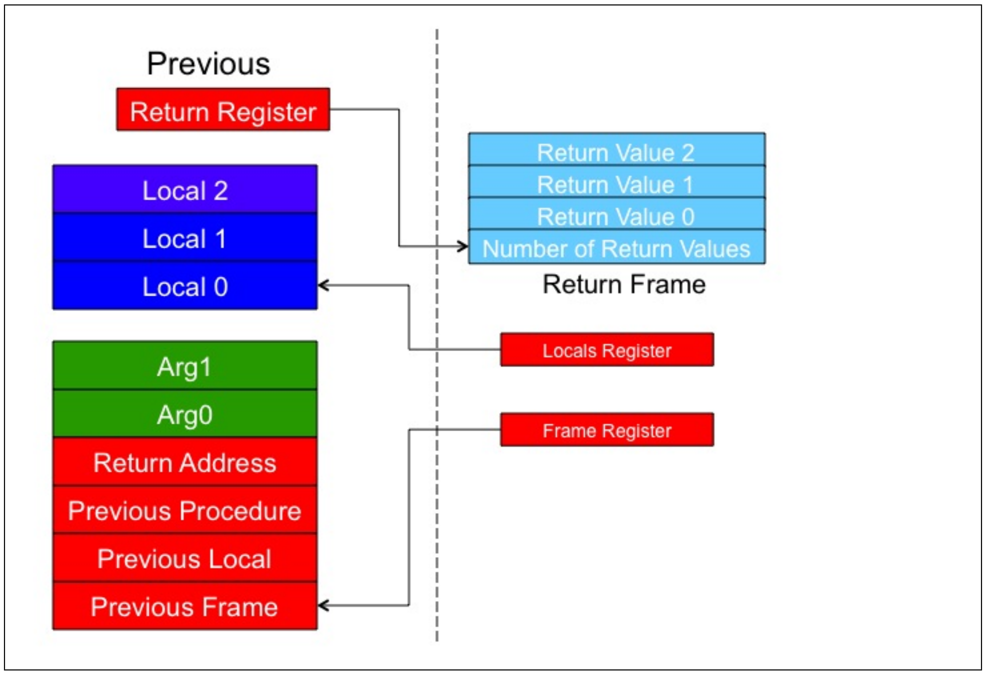


Figure 26: State After Returning

the heap and reclaimed only via garbage collection. Stack frame headers are immutable and cannot be overwritten by buffer overflows; buffer overflows cannot happen since all memory references are bounds checked; even if an overflow happens, the overwritten data will not be tagged as instructions and will not be executable (since normal code cannot manipulate tags and therefore cannot create data tagged as instructions). The stack frames use a split-stack structure, in which the caller's, callee's and return values part of the frame are distinct and are potentially in different compartments and subject to distinct access control rules.

The processor architectural state described so far includes:

1. Procedure register
2. Program counter
3. Call-Frame register
4. Locals-Frame register
5. Return-Frame register

Of these, the first four are stored in the headers of call-frames; Section 5.7.7 and Section 5.7.8 will describe other parts of the processor state that is involved in procedure calling. In particular, we will describe how environments, compartments, and principals are managed. We will also describe two more complex types of procedural objects: closures and gates. We first, however, turn to describing one further mechanism used to enforce the basic semantics of procedure call.

5.7.6 Interlocks Between Calls/Returns and Entries

One of our design goals is to provide defense in depth for the basic semantics of procedure call. So far we have seen that:

- The only non-local transfer of control is affected by procedure call and return (and trapping, which is a special kind of procedure call).

- Procedure calls can only invoke a code object
- Instructions can only live within code objects

In this section we describe one additional mechanism for enforcing these semantics. As we have said, procedure calls transfer control to an entry point of a code block and they do so via the first instruction of the code block which must be an Entry instruction. To ensure that there is no way to forge a procedure reference that would cause a non-standard transfer of control we add one simple piece of processor architectural state. As shown in Table 1, the processor holds in a special register (PREVIOUS_INSTRUCTION) the last instruction executed. As an instruction's execution begins, this register is examined; If the previous instruction was a Call instruction then, unless the current instruction is an Entry instruction, a trap is initiated.

A similar mechanism is used for Returns. If the previous instruction was a Return instruction and then unless the current instruction is a Return-Entry instruction, a trap is initiated. Notice that given these mechanisms, an attempt to branch into the middle of a code-object would lead to a trap and prevent execution of the code branched to.

Finally, we note that the normal access control rules treat instructions as immutable. Attempts to overwrite existing code lead to traps. Similarly, attempts to write code over any other data (other than uninitialized storage) is blocked. Only specific operating system components (e.g. the code loader) are allowed to write instructions into memory and they can do so only into Code-Object data structures and only if the data being overwritten has data-type tags for uninitialized storage. Collectively, these mechanisms make code injection and code hijacking attempts virtually impossible.

5.7.7 Environments and Closures

Many programming languages provide capabilities to form, store and return lexically scoped internal procedures. For example, the CommonLisp code below creates two such internal procedures, bar and baz, stores them in a list and then returns this list as its return value.

```
(defun foo (x y)
  (labels ((bar (a) (setq x (+ x y)) (+ x a))
           (baz (a) (setq x (* x y)) (* x a)))
   (list #'bar #'baz)))
```

Notice that both bar and baz refer to the variable X that is passed to foo as an argument. Each time foo is called, two new versions of bar and baz are created and returned. Notice also that bar and baz refer to the same variables X and Y and that each modifies the value of X each time it is called. On the other hand, the variable A in bar is distinct from the variable A in baz. Features similar to this go back to the earliest Lisp languages as well as to Algol 60.

The implementation techniques for supporting such language features are well known also, they involve the creation of Environments (they were called "Displays" in Algol), where an environment is typically a two dimensional object. The first dimension represents nesting level, while the second dimension represents the position of the variable within that nesting level. In the example above, the outermost nesting level corresponds to the procedure foo; this contains two slots one for X and one for Y. There are distinct second nesting levels for bar and baz, each containing one slot for A.

To implement this we introduce a new element to the processor's architectural state called the Environment Register. This points to an environment object which is essentially a vector (however, there is a distinct data type for environments). Each slot of the environment vector points to the stack frame that was in effect when an internal procedure such as bar or baz is created. In TIARA, since we have split frames, the environment vector points to a pair of a call-frame and its corresponding locals-frame.

When a procedure such as bar above is executing its arguments and local variables are held in the current call-frame and locals-frame, while those of the enclosing procedures (e.g. foo) are held by the environment. The compiler must generate appropriate instructions for referencing values in the environment for variables such as X and for referencing the call-frame (or locals-frame) for variables such as A. Figure 27 illustrates this structure. In this figure, there are two outer levels, each referencing a call-frame and locals-frame.

Notice also that because the Environment Register is a new element of the processor architectural state, the call-frame header structure includes a slot for storing the environment register of the calling procedure.

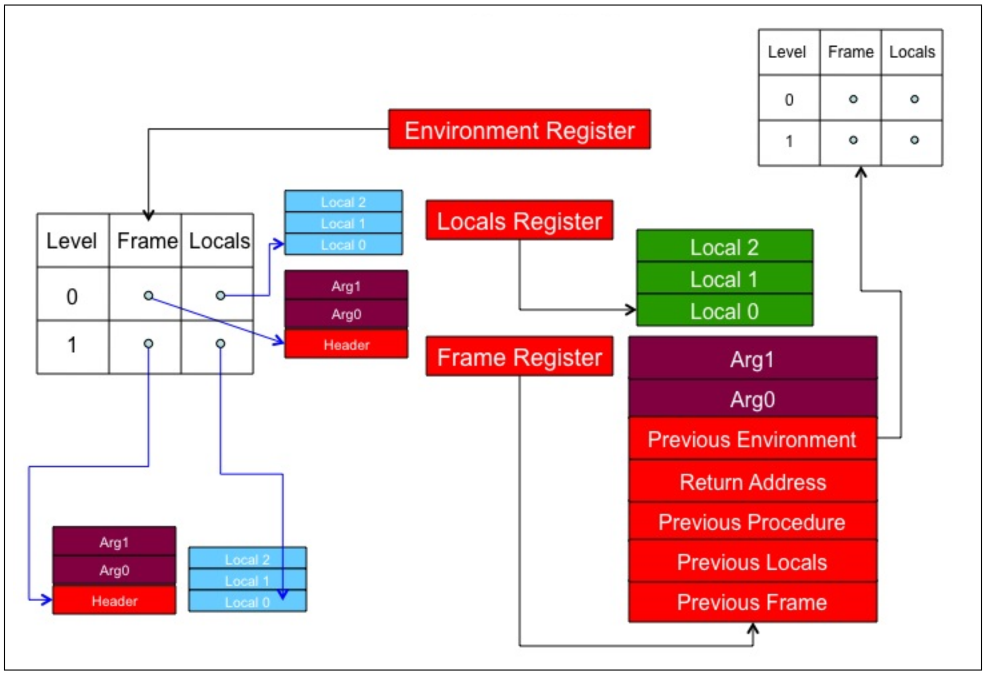


Figure 27: Environments

Earlier, we mentioned that code blocks contain a constants area and that one particularly interesting use of this area is to hold the code-blocks for internal procedures. To be precise, these code blocks do not reference the enclosing environments. Instead, when foo executes the Labels form, new data structures, called Closures are created. A closure is simply a two element vector (although closures have a unique data type); one element is the code-block while the second is an environment. The environment included in a closure is derived from the environment of the running procedure; it has one more level than the current environment, and this level points to the current call-frame and locals frame.

Thus in the example code above, we can regard foo as executing in the global environment (whatever that might be). Let us consider what happens when foo is called with actual arguments

X1 and Y1. X1 and Y1 are stored in the current call-frame and then foo begins its execution. It first allocates a new environment structure. Level 0 of this environment points to the global environment, while level 1 points to the pair of the current call-frame (including X1 and Y1) and the current locals frame. Next foo creates two closure objects, one each for bar and baz. Each of these points to the same newly created environment, but each points to the appropriate code-object from the constants area of foo. Finally a list of two elements, the two newly created closures is created and returned. Another invocation of foo with actual arguments X2 and Y2, would create a new (and different) environment referencing the global environment at level 0 and the current call-frame including X2 and Y2.

If bar or baz, in turn had internal procedures, these would be enclosed in an environment whose level 0 corresponds to the global environment, whose level 1 corresponds to the call-frame (and locals-frame) of foo when it created the closure for bar, and whose level 2 represents the call-frame (and locals-frame) of bar when it created the closure for its internal procedure.

Closures are procedural objects themselves and may be the target of a procedure-call instruction. The only difference encountered when a closure is the target of a procedure call, is that the Environment-Register is set to the environment component of the closure before control is transferred to the closure's code-block. Normal top-level procedures can be thought of (and are treated as) closures whose environment component is the global environment.

5.7.8 Principals, Compartments and Gates

In this section, we deal with the components of the processor architectural state that encodes the Principal on whose behalf the code is running and the Compartment in which storage is allocated. These are encoded in two additional processor registers, the Principal-register and the Compartment-Register. These always contain data whose object-types are references to Principals and Compartments (respectively). The full TIARA call-frame header includes slots to store the values of these registers that were current at the time of the call, as shown in Figure 28.

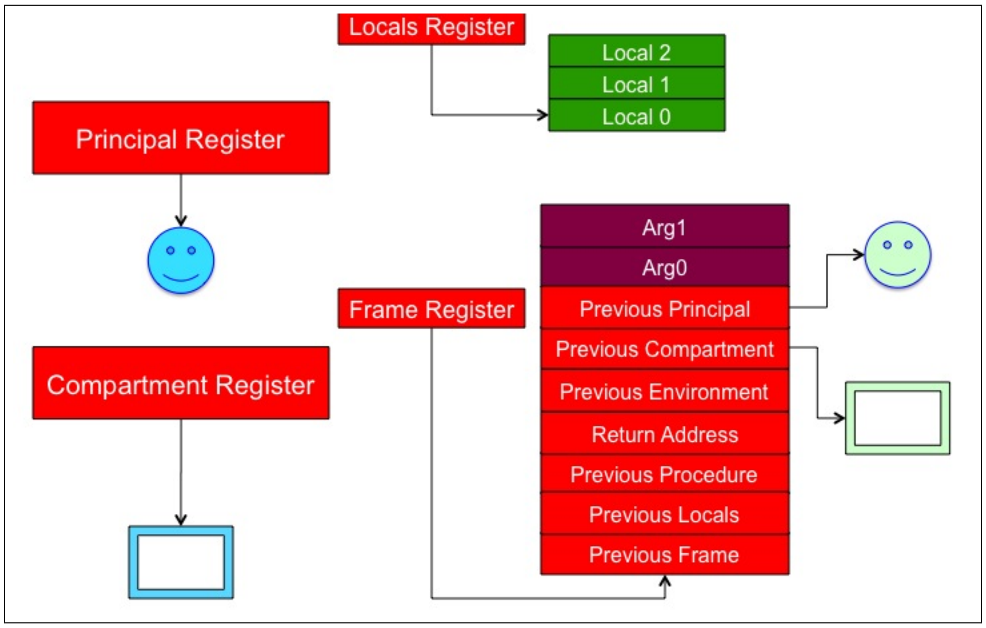


Figure 28: Principal and Compartment

In TIARA, the values of the Principal-Register and the Compartment-Register are only changed via procedure calls. This is done by calling another type of procedural object called a Gate. A gate is like a closure in that it includes a code-object and an environment; however, in addition it includes the Principal-Register and Compartment register in effect at the time of its creation as shown in Figure 29.

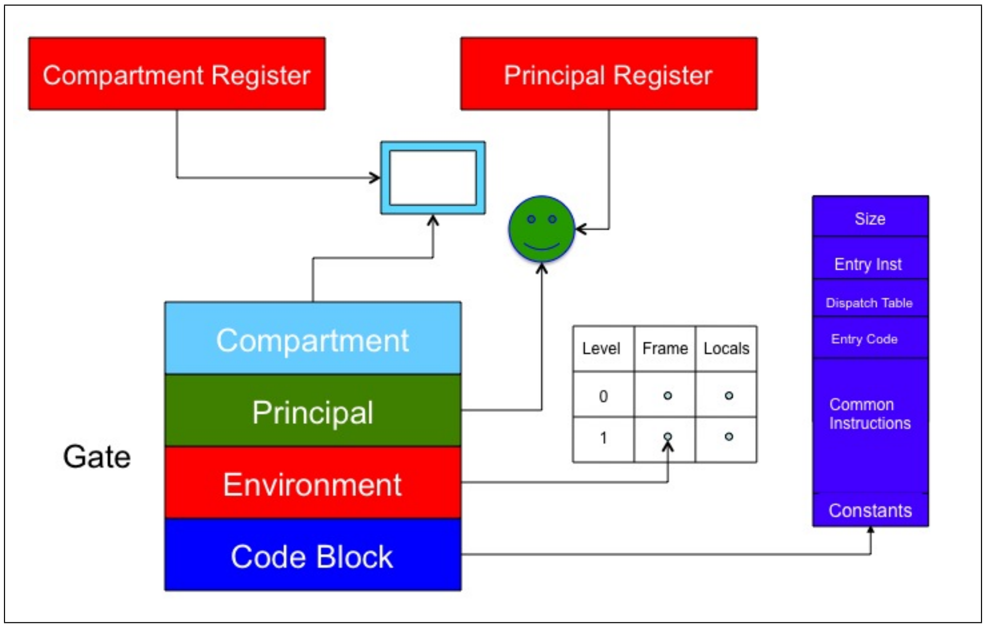


Figure 29: A Gate

When a gate is the target of a procedure call, the processor architectural state is saved in the call frame header, while the Environment-Register, Principal-Register, and Compartment-Registers are updated to those contained in the gate. Invoking a gate is the most general version of procedure calling in TIARA. The following set of figures illustrate how the processor architectural state is affected during the course of gate invocation. Figure 30 shows the full relevant architectural state at the time a new call-frame is allocated at the start of the calling process; the new call-frame is returned in Register 2 as specified by the allocate instruction.

In Figure 31, Register 1 has been loaded with a reference to a gate containing an environment, principal, compartment and code-block.

Figure 32 shows the processor architectural state in effect after that gate has been invoked. Figure 33 shows how the call-frame header contains pointers to the architectural state in effect at the time of the call. This is the state to which the processor will be returned when the gate finishes its execution and returns. The net effect of this structure is that the Compartment and Principal registers are temporarily rebound to the values in the gate during the dynamic extent of the gate's execution. Notice, that with the split stack frame structure described earlier, the Locals-Frame is allocated after the gate begins execution and is therefore allocated in the compartment of the gate, not that of the caller. Similarly, the gate executes with the privileges of its principal, not those of its caller's principal. Thus, it is possible to constrain the callee, so that it has only limited privileges to the call-frame, in particular it might have only read access to its call-fame.

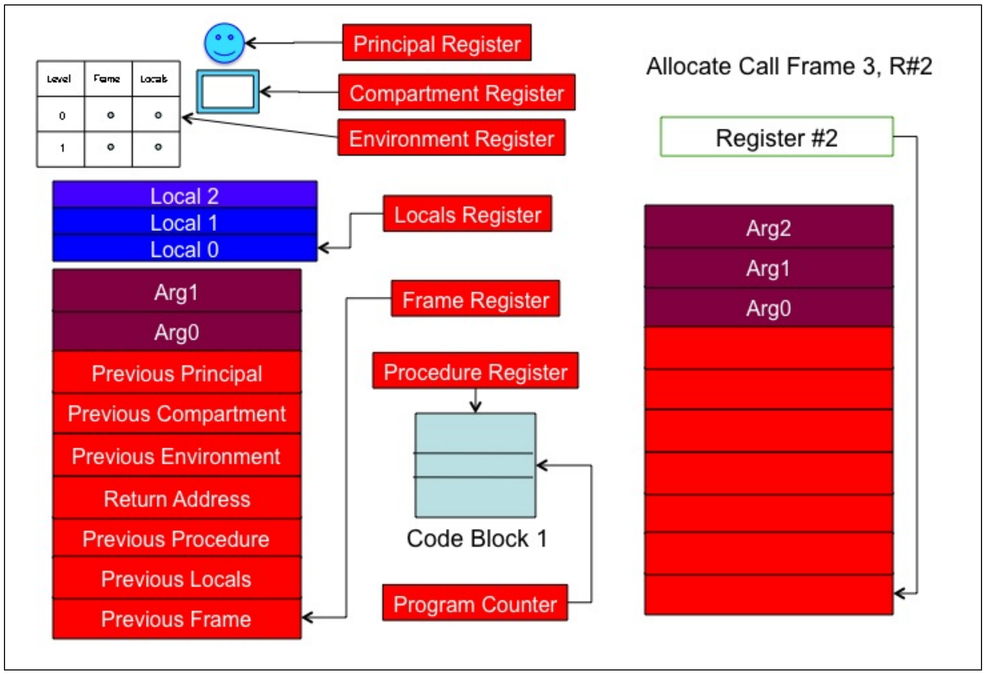


Figure 30: Allocating A Call Frame for The Call

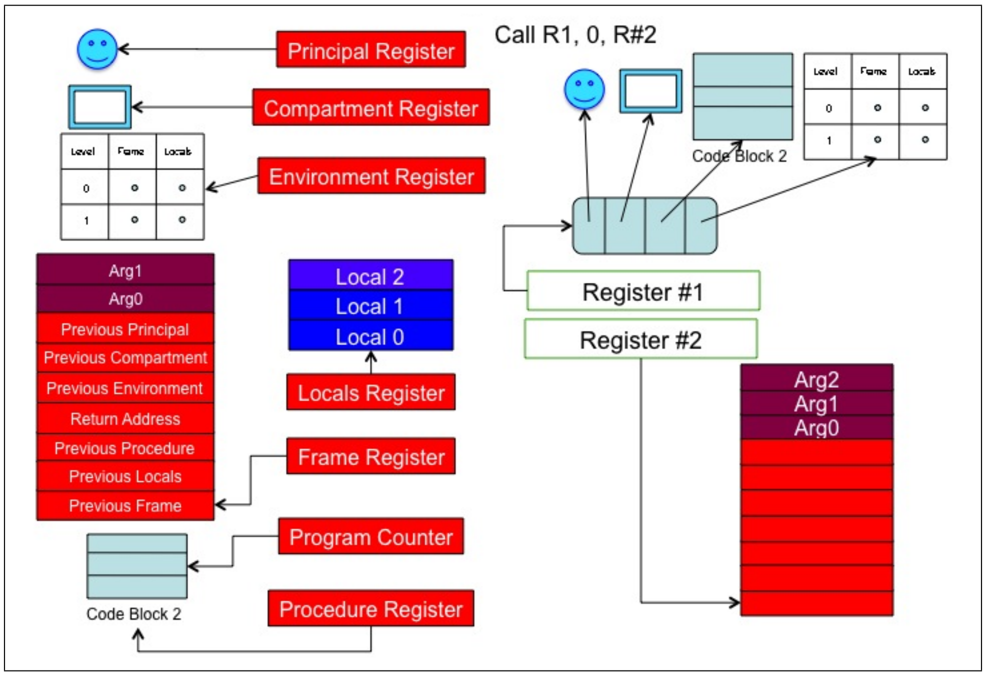


Figure 31: Executing The Call of a Gate

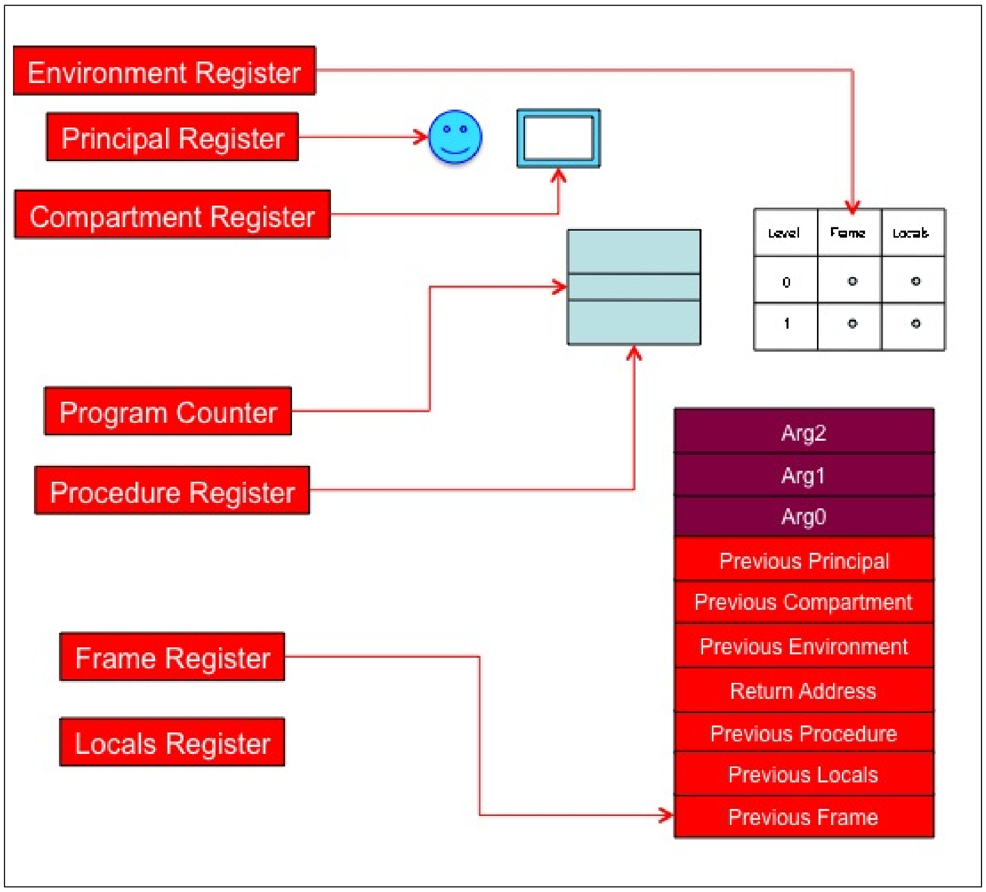


Figure 32: State After Invoking A Gate

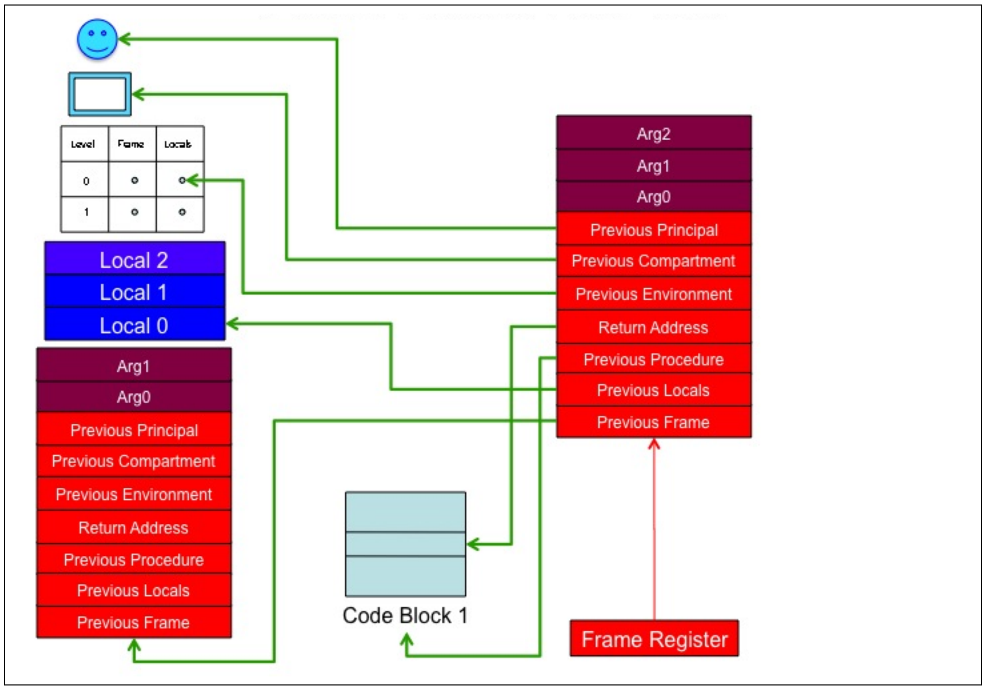


Figure 33: Frame Header After Invoking The Gate

5.7.9 Extensions

One of the novel aspects of our procedure call design is the use of heap allocated, split-frame structures. Normally such a structure would be regarded as involving high overhead by comparison to a conventional stack structure. However, our approach has the advantage that retained closures require no special handling and that there is no possibility of aliasing stack frames due to incorrect management of "frame evacuation". In this section we discuss other important capabilities for modern programming whose semantics our structure can easily support.

The first of these has to do with communications among multiple threads. One common model for this is to think Thread 1 as making a procedure call that will execute in Thread 2, where Thread 2 may be either a thread newly created to execute the procedure call or an existing thread that is called from other existing threads.

Our procedure call process involves allocating a call-frame and then executing a procedure-call instruction with the call-frame, and a gate (in the most general case, or a code-block in the simplest) as arguments. Virtually these same steps are involved in inter-thread calls. If the call is intended to create a new thread, then the call-frame is allocated and filled in just as for a normal procedure-call; however, in this case, rather than executing the procedure-call instruction, a scheduler entry-point is called with the same arguments as would be passed to the procedure-call instruction. The scheduler, in turn, builds whatever internal structures it needs to represent the thread, initialing these with the call-frame and code-object or gate passed to it. The thread is then initiated by actually executing the procedure-call once the scheduler state is updated to reflect that the new thread is in control of the processor.

A more general model is that of a server thread which accepts requests from clients running in other threads. As before, the process begins by the client allocating a call frame. This plus the procedural object (code-object or more generally a gate) is passed to the server thread; this is done by enqueuing these arguments in a process queue associated with the server thread. As each element is dequeued, a procedure-call instruction is applied to its components (call-frame and procedural object).

In these cross-thread calls, the use of gates and split frames is very important. These allow us to provide careful access controls to the called server, since it will run under a different principal than that of the caller (as was discussed earlier in the case of intra-thread calls).

So far, we have been assuming that in the cross thread calls, the called thread will simply terminate, while the calling thread keeps running (and presumably examining some element of shared state periodically if it wants to receive information back from the called thread). However, this is only one of several variants. A common variant involves the calling thread blocking until the called thread completes its request. This can be accomplished, by having the called thread allocate and initialize a return-frame which is then placed into a special location that is monitored by the calling thread. The calling thread does a busy-wait on this location and once it is filled in it executes the return instruction on the passed by return frame.

Conceptually, this looks very much like "Continuation Passing Style" programming, in which a closure (or gate) is passed to the callee for future invocation. In effect, the code after a return-entry instruction is this continuation, but we never actually make it manifest. As it happens, our current model can do a perfectly good job of supporting continuation passing style, using gates and/or closures. This may be expressed in a variety of manners at the programming language level, but the hardware primitives provided are quite adequate to support the paradigm.

There is, however, one aspect of continuation passing style which we do not yet support and that is normally regarded as a significant component of that model. This is the issue of "tail calls" in which rather than returning to its caller, a called procedure simply calls another procedure. In our current model, the call-frame and local-frame of this procedure will be retained until the called-procedure will return, because they are referenced by the call-frame header of the callee (which is in turn referenced by a processor register).

In many conventional machines, tail calls are implemented by overwriting the current frame with the data relevant to the callee and then just jumping into the code of the callee. We could adopt the same approach by not allocating a new call-frame but rather juggling the data in the current frame and then passing this same frame to the call instruction for the next procedure. However, the call frame for the next procedure might need to be a larger size than the current call-frame and this would lead to difficulties; there are possible solutions (relocating the call-frame and using forwarding pointers) but these are messy. More importantly the compartment of the next call frame might actually need to be a different compartment from that of the current call-frame.

A cleaner solution would require a new tail-call instruction. This behaves like a normal call instruction with one exception. Rather than initializing the call-frame header of the called-procedure from the processor's architectural state registers (i.e. environment, principal, compartment, call-frame, locals-frame, procedure-base and program-counter) it instead uses the values stored in the header of the current call-frame. This means that the header of the next procedure will point to the processor architectural state active when the current procedure was called. Once the callee is activated, it will not point to the current procedures call-frame and local-frame; these can then be reclaimed by the garbage collector.

6 Compiler and Instruction Set Architectures

The TIARA project focuses primarily on features required to support security, rather than on novel processor techniques intended to provide high performance. Thus, in most respects, we have chosen to support a relatively conventional instruction set. The areas that are novel concern function calling as described in Section 5.7, in particular the use of a split-frame, non-stack architecture, the use of call-entry and return-entry pairs, the provision for closures and gates. In addition to these, we also have a larger number of special purpose processor registers motivated by the need to represent principals and compartments as well as by the use of a novel frame architecture. Most importantly, the entire processor mode is object-oriented: all data is tagged with meta-data indicating their type, numbers cannot be treated as addresses, there are specific datatypes for object-references, and all memory accesses are to slots of objects, indexed by their offset from the base of the object.

6.1 Instruction Set Architecture

Most instructions are either simple register to register instructions, load-store instructions or transfer of control instructions. The register to register instructions all take 2 source operands and a destination, except for a few single source operations. Operands and destinations for these instructions are registers; when constants are required they are loaded from the constants area of the current code block. We present these using an “s-expression” notation:

```
(Instruction-name source-1 source-2 destination)
```

For example:

```
(Add 10 1 3)
```

which adds the value in register 10 to register 1 putting the result in register 3.

6.1.1 Arithmetic Instructions

Arithmetic instructions with three operands include ADD, SUBTRACT, TIMES, DIVIDE (including both generic versions that dispatch on the datatypes of the arguments, as was done in the Lisp Machines) and specific versions for each of the machine arithmetic datatypes (e.g. float, fix).

There are also two operand instructions such as MINUS, ADD-ONE.

6.1.2 Predicates

Predicates are three argument instructions: The two source arguments are the numbers of registers containing the values to compare; the destination is a register number which receives a boolean value. There is a special BOOLEAN datatype and there are two distinguished boolean values #TRUE and #FALSE. The predicate instructions include numeric predicates such as EQUAL, LESS-THEN and GREATER-THAN. These require that the arguments both be of the same numeric datatype. Two additional predicates test for identity: EQ which tests that the two arguments have the same value and datatype (but not necessarily the same compartment) and EQC (which tests that the two arguments have the same value, compartment and datatype). Because predicate instructions leave their boolean result in a register, the compiler need not place conditional branch instructions immediately following the predicate.

6.1.3 Load and Store Instructions

Load and store instructions have 3 operands, all of which are register numbers. Load instructions have the following format:

```
(Load base-register offset-register destination-register)
```

The base register must contain a value that is an object reference and the offset register must contain a value that is an integer smaller than the size of the object. The instruction tests that the offset is within bounds (and traps if it is not) adds the offset to the base and fetches the contents of that word of memory, placing it in the destination register.

The format of a store instruction is

```
(Store base-register offset-register write-data-register)
```

There is also an "immediate" version of each of these instructions in which the offset is a small integer encoded directly in the instruction.

The base address and offset registers are treated in the same manner as in the load instruction. The write-data-register contains data that is written into the appropriate word of memory.

All of the instructions mentioned so far are also checked for adherence to the access rules, and in the case of the 3-op instructions a new tag is calculated for the data placed in the destination register.

6.1.4 General and Special Purpose Registers

There are 64 general purpose registers (number 0 - 63) and 8 special purpose registers (see also Table 1).

1. Program counter. Address of the next instruction to be executed. This has data type "Program-Counter"
2. Call-Frame register. Address of the "call frame". This has datatype "call-frame-reference".
3. Locals-Frame register. Address of the "locals frame". This has datatype "locals-frame-reference".
4. Return-Frame register. Address of the "return frame". This has datatype "return-frame-reference".
5. Procedure-base register. Address of the currently executing currently executing code block. 'This has datatype "procedure-reference".
6. Environment register. Address of the current environment structure. This has data type "environment-reference".
7. Principal register. The currently executing Principal. This has datatype Principal
8. Compartment register. The compartment currently being used for data allocation. This has data type "compartment".

The call-frame, locals-frame, return-frame, procedure-base registers can all be used as base-registers in load-store instructions. None of the special registers can be directly modified by any instruction.

As described in more detail in Section 5.7, while a procedure is executing its arguments and local variables are held in two frames: The call-frame that holds arguments that are passed to procedure by its caller and the locals-frame that contains storage for local variables used by the procedure. The call-frame is allocated by the callers code while the locals frame is allocated by the callee on entry. Return-frames are allocated by a procedure to store values that are to be returned to its caller. The return-frame register is made to point to the return-frame when the callee procedure executes a return instruction. If the current procedure calls more than one subroutine, this register will be set to point to a different frame on each return. Therefore, either the return-frame register must be copied to a different register before the next call, or the values in the return-frame must be moved to the locals frame (or elsewhere) before the next call if they are still needed.

The procedure-base register points to the currently executing code-object. The code-object contains both instructions and constant values; the constant values are all stored at the end of the code-block. The compiler determines the offset to this constants area in one of its last passes, This allows it to fetch the constants by issuing a standard load instruction with the procedure-base register as the base-register.

6.1.5 Transfer of Control Instructions

Transfer of control instructions fall into two categories: Those involving local jumps within a procedure and those involved in procedure call and return. Jumps are all to offsets from the base of the current code-block, there is no ability to jump to an arbitrary address. There are both conditional and unconditional jumps; both of these have two formats: In the first, the offset from the base of the code-block is contained in a constant field within the instruction; in the second case, the offset is contained in a register designated by the instruction (which must contain a value tagged as Integer).

```
(branch register-number)
{branch-immediate offset}
```

The conditional branch instructions (branch-true, branch-false) take two arguments: 1) A register number containing one of the two boolean special values #TRUE and #FALSE and 2) the offset. These instructions also have both a normal and immediate format:

```
(branch-true condition-register-number offset-register-number)
(branch-true-immediate condition-register-number offset)
(branch-false condition-register-number offset-register-number)
(branch-false-immediate condition-register-number offset)
```

Notice that the immediate versions of these instructions contains fewer bits than can be encoded in a register. Usually, this is still more than enough to encode the full size of the code-block; however, if this isn't true, the compiler is required to load a constant from the constants area of the code block into a register and use the non-immediate version of the instruction.

6.1.6 Allocation Instructions

There is a special instruction for allocating storage. This is a 3 argument instruction. Source-1 is the datatype of the frame while Source-2 is the size of the frame. The destination is the register hardware number to receive an object reference to the newly allocated storage. The TIARA system guarantees that the newly allocated storage is initialized so that all slots of the object contain a datum whose value is 0 and whose datatype is UNINITIALIZED-STORAGE. Attempts to load a value with this datatype into memory cause a trap. There are versions of this instruction allowing either source-1 or source-2 to be immediate values as opposed to register numbers.

This instruction is used to allocate call-frames, return-frames, environment-frames, and storage for gates; they are also used to allocate normal application data structures such as arrays, classes (which are represented as instances of meta-classes in TIARA) and class instances.

There is one other instruction that allocates storage. This is the PROCEDURE-ENTRY instruction. See Section 6.1.7.

6.1.7 Procedure Call, Entry and Return Instructions

Most of the details of procedure call and return are described in Section 5.7. Here we describe the instruction formats.

The CALL instruction takes three arguments: The first is the number of the register that contains a reference to the procedure to be invoked. The second is the number of the register that contains a reference to the call-frame to be passed to the invoked procedure. The third argument is an immediate value containing the number of the entry point to be invoked.

The PROCEDURE-ENTRY instruction always appears as the first instruction in a code-block. This is a two-argument instruction and both arguments are immediate values. The first is the maximum number of entry points in the procedure. The second is the size of the procedure's Locals-Frame. The Procedure-Entry instruction allocates a locals-frame of this size and sets the Locals-Frame Register to point to the newly allocated locals-frame.

The RETURN instruction takes two arguments. The first is the number of the register that contains a reference to the return frame to be passed back to the caller. The second argument is an immediate value containing the number of return values to be returned.

The RETURN-ENTRY instruction takes no arguments. It's only purpose is to act as an interlock with the return instruction as described in Section 5.7.

6.2 Source Language

We intend for the TIARA instruction set to be general enough to support a variety of source programming languages. For the purposes of experimentation, we have chosen a language in the Lisp family, similar to CommonLisp and Scheme (as we have done elsewhere in the project). This choice was motivated but the syntactic simplicity of the language which makes it much easier to construct and modify a compiler. The source language contains the usual constructs of CommonLisp including its macro system. This makes it possible to have a very lean core language containing very few primitives that the compiler has to understand, while allowing a richer set of language features to be added using macros. Thus, we've adopted the CommonLisp LOOP syntax, because this is defined entirely in terms of macros that expand down to the core primitives.

The core primitives include: Symbolic Labels for statements, GO (unconditional branch, this is used only within macros, programmers should never use it), COND (a sequential multi-branch conditional branch), DEFUN (for defining procedures), LAMBDA (for defining unnamed internal procedures), FLET, LABELS (for defining closure, flet defines a set of closures all of which are closed in the environment surrounding the flet form, while Labels defines a set of mutually recursive closures closed in an extension of the surrounding environment) GATE (for defining named, mutually recursive internal gates), LET (for defining a local binding block), MULTIPLE-VALUE-BIND (for locally binding the results of a procedure call), RETURN and procedure applications. Catch and throw and error handling primitives should be easy to add, but we haven't explored these yet. All other language primitives are provided by macros that expand into these features.

6.2.1 Closures and Gates

The TIARA call and return architecture includes both normal procedures (e.g. those defined in the global environment) and procedure closures, i.e. locally scoped procedural objects closed in the surrounding environment. The source languages provides three forms for defining these types of objects: FLET, LABELS, GATE. Flet and Labels are used to define closures (i.e. procedures closed in the surrounding environment). GATE defines gates (i.e. extended closures that also include the current PRINCIPAL and COMPARTMENT). The syntax of FLET and LABELS is the same as in CommonLisp:

```
(defun example-1 (a b)
  (flet ((x (c) (list a c))
         (y (c) (list b c)))
    (list (x b) (y b))))
```

Here the FLET form defines two internal procedures X and Y; the variables of the enclosing environment (i.e. the variables of the procedure example-1) are visible within these internal procedures. However, neither x nor y are not visible within the scope of either x or y.

Labels allow mutual recursion:

```
(defun example-2 (a b)
  (labels ((x (c) (list (y a) c))
           (y (c) (list (x b) c)))
    (list (x b) (y b))))
```

Here both X and Y are visible within the scopes of both X and Y (because of the mutual recursion, this procedure will go into an infinite recursive descent).

The GATE form is similar to the LABELS form, but it builds gates including the enclosing environment, the current contents of the Principal and Compartment registers.

6.3 Compiler Structure

In this section we describe the structure of the TIARA compiler. We did not intend to break new ground in compiler structure in this effort, rather we intended to build a relatively simple, multiple pass compiler for a Lisp-like language. Our only intent was to deal with the new issues posed by the TIARA architecture including the call and return mechanisms, the split-frame structure describe in Section 5.7 and the use of GATES. Everything else about the compiler is relatively straightforward (and occasionally naive).

The compiler involves several passes: It first walks over the code building basic-block and variable binding-blocks. Because the syntax of our source language is an S-Expression type language, no parser is required. Rather a code recursive-descent code-walker is used to walk over the nested syntax, dispatching to routines peculiar to each language primitive as it goes. The code walker understands the syntax of each of these primitive forms and passes to the handlers information about the context in which the forms are invoked. The walker calls a handler both on entry to and exit from each form. It walks all sub-forms in appropriate order calling their entry routines after the entry routine for their parent form and their exit routines before the exit routine for their parent.

During this first pass several data-structures are built: Corresponding to each COND form, a set of basic-blocks are constructed, one for each branch of the cond. Each basic block contains a set of data-structures representing the instructions and procedure calls that will occur within the block. Forms involving function operators that correspond to instructions (e.g. + corresponds to ADD) are converted into instruction data-structures, other forms are converted into procedure call data-structures. For each form that binds variables (e.g. LET, FLET, LABELS, GATE, LAMBDA), a binding block data-structure is constructed referencing the variables bound at that level. The code-walker itself maintains a lexically nested stack of these variables. References to variables in the source code are turned into references to the variable data structures. Thus, at the end of the first pass, the source code has been converted into a set of data-structures making the control and data flows and the nested binding structure explicit.

The second pass handles the allocation of registers for intermediate temporary results. Consider the form:

```
(+ (- a b) (foo x))
```

Here the results of subtracting B from A must be held until the result of applying the function Foo to X is computed and then both results must be passed in registers to the + operation. Thus a register is temporarily allocated to the results of the subtraction (and its value preserved) until after the code for the function application. Following the function call, the result must be loaded from the return frame into a register. After these registers have been initialized the + instruction is applied to them. Also during this pass, constants that are referenced are allocated storage in the constants area of the code-block to be generated.

The third pass handles the allocation of locally bound variables to slots in the locals-frame. This could be done in a highly naive fashion in which every variable in each binding block could be allocated a unique slot in the locals frame. However, two parallel binding forms (i.e. ones not nested within one another) can share slots in the locals-frame. Instead, this pass builds a new data-structure, the frame-structure, in which the allocation for parallel binding blocks is planned. This is done by recursively descending the binding-block structures, allocating the storage for each block immediately after the storage for its parent (and therefore shared with its sibling blocks).

There is one further constraint on local-variable allocation: Variables that are referenced from within a nested closure or gate (i.e. that occur within a LABELS, FLET, or GATE form) cannot share their storage with that of parallel blocks. This is because the closure can be returned out of the procedure; the dynamic extent of the binding for these variables is different than that for normal variables whose dynamic extent follows the lexical scoping. Thus prior to allocating slots in the frame, this pass first determines all such variables and assigns them unshared storage at the beginning of the frame.

The next pass then performs a few cleanup operations: For example, it allocates slots in the local frame in which to store live variables across procedure calls. We adopt a “caller saves” approach to register saving. For example in the code below

```
(+ (- a b) (foo x))
```

assume that the result of subtracting b from a is placed into Register-0. This register is, therefore, live at the point of the procedure call to foo (i.e. the value in the register must be preserved across the procedure call and then passed to the Add instruction). Doing this requires the allocation of a new slot in the Locals-Frame. All live registers must be preserved across the call in this manner, but the locals-frame slots for this only need to be preserved across the call. This is done by introducing a new binding block including a variable for each such live register and then treating this as having the appropriate scope.

The final parts of the cleanup pass involves calculating the final location of the constants area, the maximum size of the Locals-Frame and then assigning specific slot numbers to each variable. Finally the code is generated.

One issue we haven't yet discussed is the handling of internal procedures, such as those in FLET, LABELS, GATE forms. For these the compiler is called recursively to generate a code-block for the internal procedure. All compiler states are encapsulated in a data structure, allowing this recursion. The returned code block is then allocated a slot in the constants section of the parent procedure.

Generating a call to such an internal procedure involves first constructing the appropriate closure (or gate) and then issuing a normal call. To construct a closure, the compiler must emit code to allocate a new environment that is one level deeper than the current environment and add the current Call-Frame and Locals-Frame to the last slot of this environment. Then code to allocate a closure data structure is emitted, followed by code to initialize this with pointers to the new environment, the code-block for the internal procedure, and the Principal and Compartment registers in the case of a Gate. In addition, during the compilation of the body of the internal procedure, the compiler state is initialized so that all lexically apparent enclosing variables are already in the compiler state with their appropriate frame and slot numbers already allocated.

6.4 Examples of Compiled Code

Here are examples of pairs of source code and the corresponding machine code:

```
(DEFUN TEST1 (X Y Z) (FOO (+ X 2) (BAR Y (BAZ Z) (+ X 2))))
                              (procedure TEST1)
       0                      (PROCEDURE-ENTRY 0 2)
       1                      (ALLOCATE CALL-FRAME 9 0)
       2                      (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)    X
       3                      (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 36 2)
       4                      (+ 1 2 2)
       5                      (STORE-IMMEDIATE 0 7 2)
       6                      (ALLOCATE CALL-FRAME 10 2)
       7                      (LOAD-IMMEDIATE CALL-FRAME-REGISTER 8 3)    Y
       8                      (STORE-IMMEDIATE 2 7 3)
       9                      (ALLOCATE CALL-FRAME 8 4)
      10                      (LOAD-IMMEDIATE CALL-FRAME-REGISTER 9 5)    Z
      11                      (STORE-IMMEDIATE 4 7 5)
      12                      (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 37 6)
      13                      (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 0) saved-register-0
      14                      (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 1 2) saved-register-2
      15                      (CALL 6 4 1)
      16                      (RETURN-ENTRY)
      17                      (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 1 2)  saved-register-2
      18                      (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 0 0)  saved-register-0
```

```
        19                     (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 1)
        20                     (STORE-IMMEDIATE 2 8 1)
        21                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)    X
        22                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 36 3)
        23                     (+ 1 3 3)
        24                     (STORE-IMMEDIATE 2 9 3)
        25                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 38 3)
        26                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 0) saved-register-0
        27                     (CALL 3 2 3)
        28                     (RETURN-ENTRY)
        29                     (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 0 0)  saved-register-0
        30                     (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 1)
        31                     (STORE-IMMEDIATE 0 8 1)
        32                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 39 1)
        33                     (CALL 1 0 2)
        34                     (RETURN-ENTRY)
        35                     (RETURN RETURN-FRAME-REGISTER 1)
;          -- constants area --
        36                     (constant FIXNUM 2)
        37                     (constant PROCEDURE-REFERENCE BAZ)
        38                     (constant PROCEDURE-REFERENCE BAR)
        39                     (constant PROCEDURE-REFERENCE FOO)


(DEFUN TEST2 (X Z) (COND ((MUMBLE X) (+ X 2)) (X (+ X 3)) (T (PRINT X) X)))

                               (procedure TEST2)
         0                     (PROCEDURE-ENTRY 0 0)
         1     branch-7783     (ALLOCATE CALL-FRAME 8 1)
         2                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 2)    X
         3                     (STORE-IMMEDIATE 1 7 2)
         4                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 29 3)
         5                     (CALL 3 1 1)
         6                     (RETURN-ENTRY)
         7                     (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 1)
         8                     (BRANCH-FALSE 1 13)
         9                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)    X
        10                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 30 2)
        11                     (+ 1 2 0)
        12                     (BRANCH 26)
        13     branch-7784     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 0)    X
        14                     (BRANCH-FALSE 0 18)
        15                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 31 1)
        16                     (+ 0 1 0)
        17                     (BRANCH 26)
        18     branch-7786     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 32 1)
        19                     (BRANCH-FALSE 1 26)
        20                     (ALLOCATE CALL-FRAME 8 1)
        21                     (STORE-IMMEDIATE 1 7 0)
        22                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 33 2)
        23                     (CALL 2 1 1)
        24                     (RETURN-ENTRY)
        25                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 0)    X
        26     next-7785       (ALLOCATE RETURN-FRAME 1 0)
```

```
       27                      (STORE-IMMEDIATE 0 0 0)
       28                      (RETURN 0 0)
;          -- constants area --
       29                      (constant PROCEDURE-REFERENCE MUMBLE)
       30                      (constant FIXNUM 2)
       31                      (constant FIXNUM 3)
       32                      (constant SYMBOL T)
       33                      (constant PROCEDURE-REFERENCE PRINT)
```

```
(DEFUN LET-TEST-3 (X) (LET ((A 1) (B X)) (+ A B)))
                               (procedure LET-TEST-3)
         0                     (PROCEDURE-ENTRY 0 2)
         1                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 9 0)
         2                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 0) A
         3                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)    X
         4                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 1 1) B
         5                     (+ 0 1 2)
         6                     (ALLOCATE RETURN-FRAME 1 3)
         7                     (STORE-IMMEDIATE 3 0 2)
         8                     (RETURN 3 0)
;          -- constants area --
         9                     (constant FIXNUM 1)
```

```
(DEFUN LOOP-TEST (J) (LOOP FOR I BELOW J DO (PRINT I)))
                               (procedure LOOP-TEST)
         0                     (PROCEDURE-ENTRY 0 2)
         1                     (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 0)    J
         2                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 0) G7844
         3                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 21 1)
         4                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 1 1) I
         5      NEXT-LOOP
                branch-7845    (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 1 0)  I
         6                     (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 0 1)  G7844
         7                     (>= 0 1 2)
         8                     (BRANCH-FALSE 2 10)
         9                     (BRANCH 19)
        10      next-7847      (ALLOCATE CALL-FRAME 8 2)
        11                     (STORE-IMMEDIATE 2 7 0)
        12                     (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 22 3)
        13                     (CALL 3 2 1)
        14                     (RETURN-ENTRY)
        15                     (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 1 0)  I
        16                     (1+ 0 1)
        17                     (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 1 1) I
        18                     (BRANCH 5)
        19      END-LOOP       (ALLOCATE RETURN-FRAME 0 0)
        20                     (RETURN 0 0)
;          -- constants area --
        21                     (constant FIXNUM 0)
        22                     (constant PROCEDURE-REFERENCE PRINT)
```

```
(DEFUN LABELS-TEST-1 (A B) (LABELS ((FOO (B) (LIST A B)) (BAR (A) (LIST (FOO A) B))) (BAR A)))
                              (procedure LABELS-TEST-1)
         0                    (PROCEDURE-ENTRY 0 2)
         1                    (ALLOCATE ENVIRONMENT-FRAME 3 0)
         2                    (STORE-IMMEDIATE 0 0 CALL-FRAME-REGISTER)
         3                    (STORE-IMMEDIATE 0 1 LOCALS-FRAME-REGISTER)
         4                    (STORE-IMMEDIATE 0 2 ENVIRONMENT-FRAME-REGISTER)
         5                    (ALLOCATE CLOSURE-FRAME 2 2)
         6                    (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 1 2) BAR
         7                    (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 23 1)
         8                    (STORE-IMMEDIATE 2 0 1)
         9                    (STORE-IMMEDIATE 2 1 0)
        10                    (ALLOCATE CLOSURE-FRAME 2 3)
        11                    (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 3) FOO
        12                    (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 24 1)
        13                    (STORE-IMMEDIATE 3 0 1)
        14                    (STORE-IMMEDIATE 3 1 0)
        15                    (ALLOCATE CALL-FRAME 8 0)
        16                    (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)    A
        17                    (STORE-IMMEDIATE 0 7 1)
        18                    (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 1 4)  BAR
        19                    (CALL 4 0 1)
        20                    (RETURN-ENTRY)
        21                    (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 0)
        22                    (RETURN RETURN-FRAME-REGISTER 1)
;          -- constants area --
        23                    (constant PROCEDURE-REFERENCE (INTERNAL BAR LABELS-TEST-1))
        24                    (constant PROCEDURE-REFERENCE (INTERNAL FOO LABELS-TEST-1))

                              (procedure (INTERNAL BAR LABELS-TEST-1))
         0                    (PROCEDURE-ENTRY 0 1)
         1                    (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 0 0)
         2                    (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 1 1)
         3                    (ALLOCATE CALL-FRAME 9 2)
         4                    (ALLOCATE CALL-FRAME 8 3)
         5                    (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 4)    A
         6                    (STORE-IMMEDIATE 3 7 4)
         7                    (LOAD-IMMEDIATE 1 0 5)                      FOO
         8                    (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 2) saved-register-2
         9                    (CALL 5 3 1)
        10                    (RETURN-ENTRY)
        11                    (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 0 2)  saved-register-2
        12                    (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 3)
        13                    (STORE-IMMEDIATE 2 7 3)
        14                    (LOAD-IMMEDIATE 0 8 3)                      B
        15                    (STORE-IMMEDIATE 2 8 3)
        16                    (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 20 4)
        17                    (CALL 4 2 2)
        18                    (RETURN-ENTRY)
        19                    (RETURN RETURN-FRAME-REGISTER 1)
;          -- constants area --
        20                    (constant PROCEDURE-REFERENCE LIST)
```

```
                          (procedure (INTERNAL FOO LABELS-TEST-1))
        0                 (PROCEDURE-ENTRY 0 0)
        1                 (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 0 0)
        2                 (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 1 1)
        3                 (ALLOCATE CALL-FRAME 9 2)
        4                 (LOAD-IMMEDIATE 0 7 3)                     A
        5                 (STORE-IMMEDIATE 2 7 3)
        6                 (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 4)   B
        7                 (STORE-IMMEDIATE 2 8 4)
        8                 (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 12 5)
        9                 (CALL 5 2 2)
       10                 (RETURN-ENTRY)
       11                 (RETURN RETURN-FRAME-REGISTER 1)
;          -- constants area --
       12                 (constant PROCEDURE-REFERENCE LIST)

(DEFUN FLET-TEST-1 (A B) (GATE ((FOO (B) (LIST A B))) (FOO A)))
                          (procedure FLET-TEST-1)
        0                 (PROCEDURE-ENTRY 0 1)
        1                 (ALLOCATE ENVIRONMENT-FRAME 3 0)
        2                 (STORE-IMMEDIATE 0 0 CALL-FRAME-REGISTER)
        3                 (STORE-IMMEDIATE 0 1 LOCALS-FRAME-REGISTER)
        4                 (STORE-IMMEDIATE 0 2 ENVIRONMENT-FRAME-REGISTER)
        5                 (ALLOCATE GATE-FRAME 4 2)
        6                 (STORE-IMMEDIATE LOCALS-FRAME-REGISTER 0 2) FOO
        7                 (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 20 1)
        8                 (STORE-IMMEDIATE 2 0 1)
        9                 (STORE-IMMEDIATE 2 1 0)
       10                 (STORE-IMMEDIATE 2 2 PRINCIPAL-REGISTER)
       11                 (STORE-IMMEDIATE 2 3 COMPARTMENT-REGISTER)
       12                 (ALLOCATE CALL-FRAME 8 0)
       13                 (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 1)   A
       14                 (STORE-IMMEDIATE 0 7 1)
       15                 (LOAD-IMMEDIATE LOCALS-FRAME-REGISTER 0 3)  FOO
       16                 (CALL 3 0 1)
       17                 (RETURN-ENTRY)
       18                 (LOAD-IMMEDIATE RETURN-FRAME-REGISTER 0 0)
       19                 (RETURN RETURN-FRAME-REGISTER 1)
;          -- constants area --
       20                 (constant PROCEDURE-REFERENCE (INTERNAL FOO FLET-TEST-1))

                          (procedure (INTERNAL FOO FLET-TEST-1))
        0                 (PROCEDURE-ENTRY 0 0)
        1                 (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 0 0)
        2                 (LOAD-IMMEDIATE ENVIRONMENT-REGISTER 1 1)
        3                 (ALLOCATE CALL-FRAME 9 2)
        4                 (LOAD-IMMEDIATE 0 7 3)                     A
        5                 (STORE-IMMEDIATE 2 7 3)
        6                 (LOAD-IMMEDIATE CALL-FRAME-REGISTER 7 4)   B
        7                 (STORE-IMMEDIATE 2 8 4)
        8                 (LOAD-IMMEDIATE PROCEDURE-BASE-REGISTER 12 5)
        9                 (CALL 5 2 2)
       10                 (RETURN-ENTRY)
       11                 (RETURN RETURN-FRAME-REGISTER 1)
```

```
;           -- constants area --
        12                    (constant PROCEDURE-REFERENCE LIST)
```

7 Upper Level Application Infrastructure

The TIARA project has involved work at the hardware level (STA), the operating system (ZKOS), and at the application middleware level. Each of these levels provides a set of guarantees to the layers above it. This makes it possible to give strong assurances that the controls at each level cannot be bypassed. For example, the tag management unit at the STA level allows us to make strong guarantees that the separation properties we desire at the ZKOS level cannot be violated. Similarly, the guarantees provided by STA plus ZKOS make it possible to guarantee that the core facilities at the next level are themselves inviolable.

In the following sections we will discuss three facilities provided at these upper levels. The first of the facilities provides for the erection of application level access controls. These are similar to the protections provided by the STA hardware; however, these operate at a more macroscopic level, erecting controls around the invocation of methods rather than around individual instructions. Access controls provide a first layer of defense against attacks and misuse of the system; furthermore, this layer is relatively inexpensive. The second facility provides a second line of defense, execution monitoring, which guarantees that the behavior of an application does not violate the invariants of an architectural model of that application. This provides a more extensive set of protections, guaranteeing that data and control flows as well as input-output invariants of all methods are adhered to; however, it does so with greater computational cost. This facility is based largely on prior work on the AWDRAT system [68] (conducted as part of the DARPA sponsored SRS program). We have modified the AWDRAT code so that it works in the context of the TIARA infrastructure (as opposed to the Java environment considered in the SRS project). The third facility provides for the dynamic construction and logging of computational traces that explain how each value in an application was computed.

These facilities are all implemented using a common wrapper technology provided as part of the TIARA object system. This is conceptually similar to the object system of CLOS [33, 12]; in particular, the TIARA object system provides for a form of method combination in which wrappers related to the three facilities just listed gain control before normal methods. This allows the wrappers to gather information, check for whether behavior is proceeding as expected, enforce access control rules, and build computational traces. The general scheme is shown in Figure 34.

We will first describe the access control facility in Section 7.1; this section will also provide more detail on the wrapper system used to implement all three facilities. We also describe how this level of access control is used to recursively provide protections against the modification of the the wrappers used to implement the controls. However, because of general strategy of defense in depth, we also use the controls provided by the STA hardware and the isolation provided by ZKOS to prevent any instruction from modifying the data structures implementing the object system and its wrappers. Thus, we have a high degree of confidence that these upper-level facilities cannot be bypassed either through attacks operating at the level of the operating system or by attacks that depend on the much more difficult task of code injection.

We will then describe in Section 7.2 the implementation of the execution monitoring facility; this is described as the Plan Layer in Section 5. We will not spend much time on this component since the design is similar to that described in [68]. Finally we will describe in Section 7.3 the facility responsible for building computation trees that describe how individual values were computed; this is referred to as the Data Accountability layer in Section 2.5.

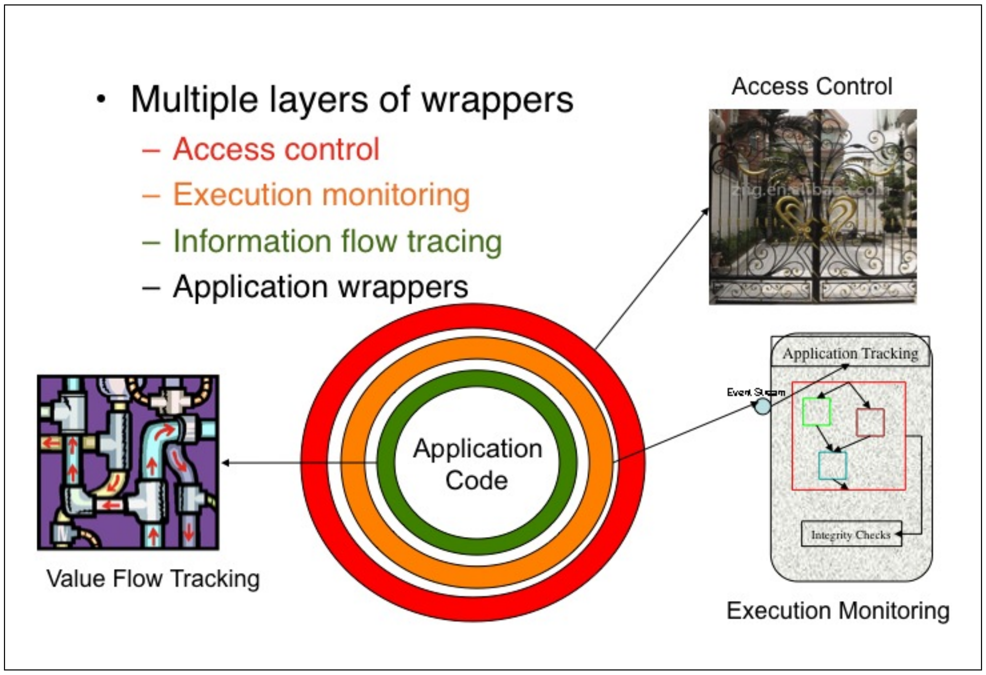


Figure 34: Wrappers Implement TIARA's Upper Level Software Infrastructure

7.1 Implementing Application Access Controls

Access control in TIARA is implemented at more than one level. The hardware, in particular the Tag Management Unit implements extremely fine grained access controls; it enforces hardware level rules expressing access right to individual words in memory (or registers) with checking being performed on every instruction.

There is also a role for access controls to be imposed at a higher level of abstraction and acting more macroscopically. Since TIARA is a thoroughly object-oriented system, the obvious level at which to erect these capabilities is at the granularity of individual objects with checking being performed at each method invocation.

In this section, we report on the development of such a model and its implementation in CommonLisp. We selected CommonLisp as the vehicle for this experiment for both principled and pragmatic reasons. Pragmatically, it is a language with which we are familiar and that is readily available. CommonLisp also has many of the properties that we would like for a language used in TIARA implementations: It has a very general model of object-oriented programming involving multiple inheritance and multi-method dispatch. It also provides a meta-level that is useful for exploring implementations of extended object models. This extreme flexibility is a double-edged sword: it gives us the power to easily build highly dynamic, self-reflective systems, but it also raises the question of whether this power can be contained.

This section proceeds as follows: First we describe the access control that we wish to impose at this level. Essentially, this is the object-oriented view of the model imposed by the hardware, involving principals, compartments, gates and access rules. Next we show an example of how this model can be used to build a simple example of a TIARA application in which various objects share data in controlled ways. We also illustrate how these techniques are used in the trial implementations of ZKOS. We then describe the techniques used to implement this model in CommonLisp; these are suggestive of the techniques that would be used in an ultimate TIARA system. One important aspect of our model is that it provides for a dynamic object-oriented system in which classes and methods can be added or redefined as the system runs. This raises the question of how to express access rules that control this dynamism and regulate who is allowed to change which methods. We show that, in particular, these access rules have the power to protect themselves, guaranteeing that the chain of trust cannot be unraveled. We then consider what we learned from the CommonLisp implementation, in particular we consider what features of CommonLisp are impediments to achieving our goals.

7.1.1 The Model

Access control fundamentally is concerned with specifying who is allowed to do what to which objects. Conceptually, this can be expressed using an access control matrix [35, 36]. The most common use of this model is the use of access control lists on files in commodity operating systems; these control the granting of a limited set of privileges (*e.g.* read, write, execute) to specific entities (*i.e.* users or groups) over coarse grained objects (*files, directories*). Such a model is at the least inconvenient and inexpressive. Later models [5] enrich the language by including notions such as hierarchies (or Directed Acyclic Graphs) of users, objects, and operations while [26, 9] introduce the notion of the roles a user might be playing and managing access in terms of such roles.

Our model draws on these; however, we are concerned with the systematic fine-grained control of access to *all* objects within the memory of the system, not just to external objects like files or directories. Thus, we center our model around CommonLisp objects and the generic-functions that operate on them. However, we extend this model as follows:

- **Objects**: Every object is an instance of a class. In addition, every object "lives" within a compartment.
- **Compartments**: A compartment is an aggregation of objects whose access rights are managed in common. Every object belongs to a single compartment. Compartments are themselves represented as objects and are therefore located within the class hierarchy.
- **Principals**: A principal is any active entity within the system, such as a user or a system components (*e.g* the scheduler). Principals are objects and therefore fit within the class system and live within a compartment.
- **Threads**: Every thread has an associated principal on whose behalf the thread is executing. Every thread also has an associated compartment in which it allocates new objects.
- **Access Rules**: An access rule controls which principals are allowed to invoke which generic functions on objects in which compartments. An access rule is specified in a manner similar to a method, it contains:
  - The name of a generic function.
  - A class specifier for the principal.
  - A class specifier for the compartment of each argument.

  An access rule is applicable when:

  - The principal of the current thread is a member of the class specified by the principal class specifier.
  - The compartment of each actual argument is a member of the class specified by the compartment class specifier for the corresponding formal argument.

  The "body" of an access rule is limited to the keywords :permitted and :denied.

  Figure 35 shows an example access rule. This rule states that any Principal whose class is "Demo-Principals" can perform a Plus operation on any pair of operands both of which are in "Demo Compartments".

  When a generic function is applied to a set of arguments, the compartments of the arguments are used to fetch all applicable access rules. These are processed in most specific-first order, looking for an access rule whose body is :permitted or :denied. If an access rule with a :permitted body is found first, the thread is allowed to invoke the generic function on the arguments. If an access rule with a :denied body is found first, then an error is signaled. Finally, if there is no applicable access rule then an error is signaled. Thus, the default behavior is to deny access to any generic function (the default is to fault).

- **Gates**: A gate is a package of a generic-function, a compartment and a principal. A gate is a funcallable object; when it is called, the principal and compartment of the thread are rebound to those of the gate and the generic-function is called. The principal and compartment are rebound on return (normal or abnormal) from the generic function. These are very similar to the gates in MULTICS [51]. Gates are the only means for changing a thread's compartment and principal.

  Gates are objects and therefore live in a compartment. Access rules also control which principals can invoke which gates (based on the compartments of the gate and of the arguments).

```
(def-aif-method plus :permitter ((principal demo-principals)
 (p1 demo-compartments)
 (p2 demo-compartments))
  :permitted)
```

Figure 35: An Access Rule

We make the (invalid for CommonLisp) assumption that all operations are generic functions. As a result, the ability to allocate, access, and modify all objects is controlled by access rules.

In particular, one class of access rules control read/write methods. These methods specify which principals can access which slots of objects in a particular compartment.[11]

A second class of access rules controls allocation of objects which is possible since the MOP specifies a series of generic-functions (*e.g.* allocate-instance, initialize-instance) that constitute the implementation of make-instance. First of all these rules control whether the principal of the running

[11]In CommonLisp slots can also be accessed using the function slot-value. The MOP specifies that slot-value is defined in terms of the generic-function slot-value-using-class. Although we have not done so yet, the access rules for reader/writer methods should also control slot-value in exactly the same way.

thread is attempting to allocate the object in the current compartment of that thread and secondly checks whether the principal is allowed to allocate in that compartment at all. Finally, these rules check that the principal has the right to initialize each of the slots as specified.

A third class of access rules specify which gates can be invoked by which principals.

Finally, access rules can be specified for arbitrary generic functions limiting a principal's ability to invoke specific services.

There are several consequences of the use of such access rules. Suppose an object is in compartment-1 and that principal-2 is not sanctioned to access the slots of objects in compartment-1. Then from the point of view of principal-2, object-1 is opaque; even if its slots contain objects that principal-2 can manipulate, principal-2 cannot discover these objects via object-1.

Another consequence is that privilege in the system need not be hierarchical; there need be no single actor (like the kernel or "root" user of an OS) that has all privileges. Each principal has a limited set of privileges, ideally as few as necessary to do its job. Inheritance can be used to compactly specify classes of principals that share privileges and classes of compartments whose privileges they share; but this need not imply strictly hierarchical layers of increasing privilege.

Information flows can be effected only by reading information from objects in one compartment and then writing this information into objects in another compartment. But, because, these actions are governed by access rules it should be possible, in principle, to formally analyze the information flows sanctioned.

Since the only way to change a thread's privileges is to invoke a gate and since gates can only be created if the access rules sanction the allocation and initialization operations involved, at least in principle it is possible to formally analyze what privileges are accessible to a thread.

Thus, as long as the set of access rules is static, predictable control of information flow is possible, even in a highly dynamic environment like CommonLisp.

7.1.2 An Application Example

In this section we describe a simple application that uses these building blocks. The application is a simplifed version of a graphical editor for military "couse of action diagrams" that we had built as part of a previous project. The tool is known as CCOAT (Commander's Course of Action Tool). Our goal was to illustrate that we could simply retrofit access controls into this existing body of code. In the retrofit, there are four compartments of data:

1. Top: Data in this compartment is privileged.

2. Blue: Data in this compartment represents the "blue view" of the situation.

3. Red: Data in this compartment represents the "red view" of the situation.

4. Common: Data in this compartment is open and accessible to all.

There are three types of users, each with different access rights:

1. Commanders: Are able to create, modify and access all application data.

2. Blue: Are able to create and modify data in the Blue compartment and read data in the Common compartment. Blue users cannot access or modify data in the Top or Red compartments. Blue users execute with the Blue compartment as their default compartment.

3. Red: Are able to create and modify data in the Red compartment and read data in the Common compartment. Red users cannot access or modify data in the Top or Blue compartments. Red users execute with the Red compartment as their default compartment.

Figure 36 illustrate the code used to establish the compartments and principals. This involves nothing more than creating sub-classes of the base classes **User-Principals** and **User-Compartments**, and then instantiating instances of these classes. The class for the Common compartment has both the Red and Blue compartment as super-classes; this allows principals with access to Red compartments to inherit access to the Common compartment.

```
;;  Principals
(defclass ccoat-principals (user-principals) ())
(defclass ccoat-commander-principals (ccoat-principals) ())
(defclass ccoat-blue-principals (ccoat-principals) ())
(defclass ccoat-red-principals (ccoat-principals) ())

(defclass ccoat-compartments (user-compartments) ())
(defclass ccoat-commander-compartments (ccoat-compartments) ())
(defclass ccoat-blue-compartments (ccoat-compartments) ())
(defclass ccoat-red-compartments (ccoat-compartments) ())
(defclass ccoat-common-compartments
  (ccoat-red-compartments ccoat-blue-compartments)
   ())

(defparameter *ccoat-commander-compartment*
    (make-instance 'ccoat-commander-compartments
      :name "ccoat commander compartment"
      :magic-number (incf *user-compartment-number*)))
...
```

Figure 36: Code Establishing Compartments and Principals in CCOAT

Next we need to define our data structures and to specify which operations on these objects are available to which principals. Since the application is a graphical editor, most application objects are instances of sub-classes of a base class called **point-like-objects**. Figure 37 shows the code necessary to extend the appropriate access rights to these objects for the three types of principals. To make the specification more compact, we created two macros **Extend-Multiple-Read-Permissions** and **Extend-Multiple-Write-Permissions**. We show the expansion of this macro for the Commander class but use the macros for the other classes.

Other data structures and access rules are specified in a similar manner. The CCOAT editor is implemented as a Common Lisp Interface Manager (CLIM) application. Separate CLIM application frames are set up for each user. The application provides commands to create objects representing different types of military units. Red users can only create red objects while blue users can only create blue object. Commanders can create objects in any compartment. The background map is created in the common compartment at application startup. All application objects are held in a common data structure that is shared by all users. The CLIM display loop in each application frame iterates over the objects in this common data structure, presenting each on its display. However, when a red user attempts to display a blue object (or a blue user attempts to display a red object)

```
(defclass point-like-object ()
  ((x :initarg :x :accessor x)
   (y :initarg :y :accessor y)))

;;; commanders can see anything
(defmethod x :permitter
  ((principal ccoat-commander-principals)
   (point-like-object ccoat-compartments))
  t)
(defmethod y :permitter
  ((principal ccoat-commander-principals)
   (point-like-object ccoat-compartments))
  t)

;;; commanders can change anything
(extend-multiple-write-permissions (x y)
 ((new-value t)
  (principal ccoat-commander-principals)
  (point-like-object ccoat-compartments)))

;;; red can mung red
(extend-multiple-write-permissions  (x y)
 ((new-value t)
  (principal ccoat-red-principals)
  (point-like-object ccoat-red-compartments)))
```

```
;;; blue guys can see blue data
(extend-multiple-read-permissions  (x y)
 ((principal ccoat-blue-principals)
  (point-like-object ccoat-blue-compartments)))
;;; red guys can see red data
(extend-multiple-read-permissions (x y)
 ((principal ccoat-red-principals)
  (point-like-object ccoat-red-compartments)))

;;; blue can mung blue
(extend-multiple-write-permissions (x y)
 ((new-value t)
  (principal ccoat-blue-principals)
  (point-like-object ccoat-blue-compartments)))
```

Figure 37: CCOAT Access Rules

an access violation is signalled. This is because displaying the object requires accessing its **X** and **Y** slots and this is prohibitted by the access rules. The CLIM display loop catches and ignores this access violation and then goes on to display the next object. The net result is that the Red user's display shows only the background and red objects, while the blue user's display shows only the background blue objects. The commander's display shows all objects. Figure 38 shows an example of a set of CCOAT displays.

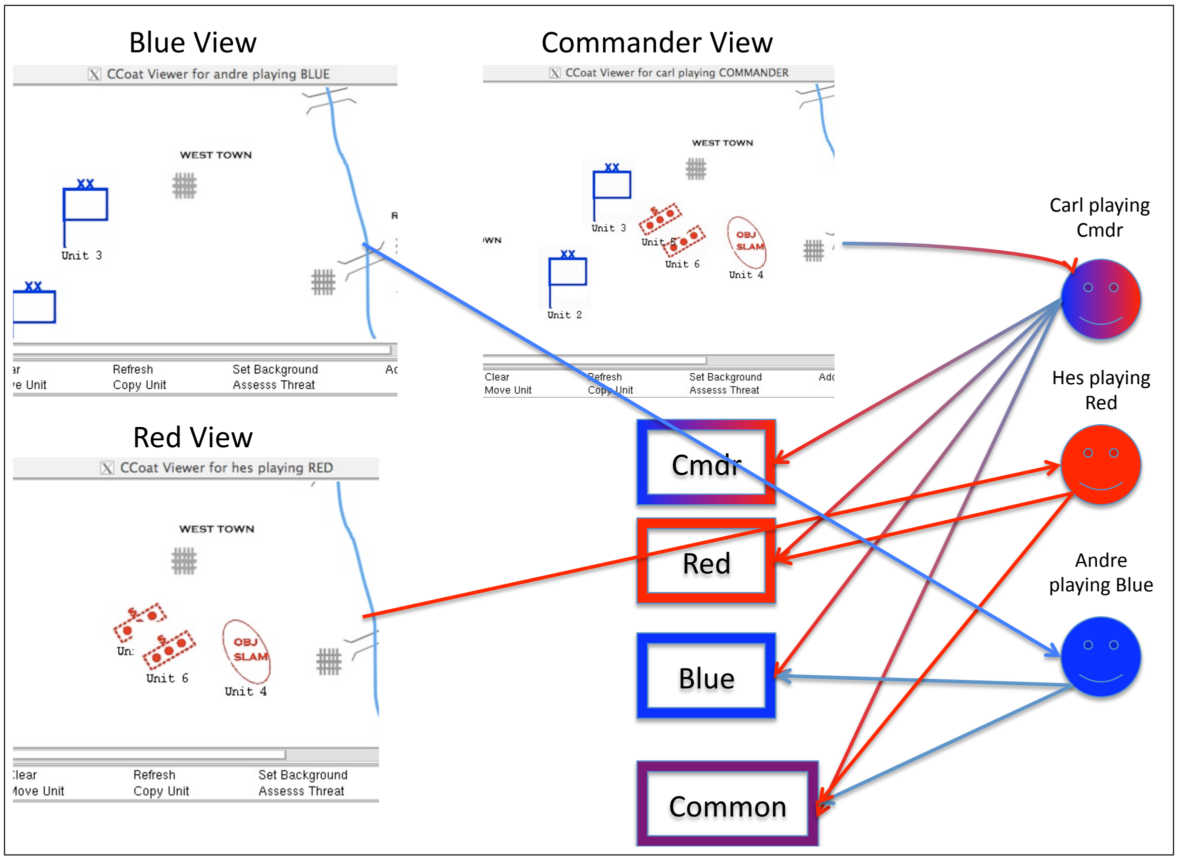


Figure 38: An Example CCOAT Display

7.1.3 Using the Model to Build Secure Components

In the previous section we illustrated how the building blocks described in Section 7.1.1 can be used to build applications that segregate data into multiple compartments, extending different access rights to different classes of principals. In this section, we discuss how these building blocks allow one to build operating system software in a novel way that is mindful of the need to maintain security properties such as privacy and integrity. We are guided by principles described by Saltzer and Schroeder in their retrospective review of the Multics experience [59]. These principles include:

1. *Complete mediation.*
2. *Least privilege.*
3. *Separation of privilege.*

Current computer systems violate these principles with abandon. Operating systems, for example, divide the world into a "kernel" that runs with unlimited privileges and "user spaces" that are isolated from one another and the kernel. However, within each user space there are few tools for decomposing privilege and for guaranteeing that all operations are, in fact, checked for authority. Complicating the problem is that the kernel in most systems has grown to enormous size offering a large "attack surface" with the attraction for attackers of achieving full control of a system. Much server software, even those that run in user mode, present the same problem; a large complex body of software manages many users' data without adequate controls on access and information flow.

Compartments, principals, access rules and gates provide a set of building blocks for a different way of structuring such software. To illustrate the process, and the design patterns that emerge, we will use an example of a "log manager" a utility that accepts log entries from a variety of sources and commit these to persistent storage (presumably in encrypted form). The challenge is to build such a utility in such a way that there is a very low possibility that it will leak information between its various clients.

The starting point of the design is to consider who are the actors in the scenario, what data these actors manipulate, what interactions between the actors are required and what constraints exist on information flow. From these one then defines a set of principals corresponding to the actors and compartments into which the data is aggregated. In the current example, we create a principal for the Log-Manager *per se* and a principal for each client (*e.g.* User-1 and User-2); we also create a compartment for each of the actors (*e.g.* Log-Manager-Compartment, User-1-Compartment, User-2-Compartment). The access rules specify that the Log-Manager can access data in the Log-Manager-Compartment and that User-1 can access any data in Compartment-1 in any way desired (and similarly for User-2 and Compartment-2). At this point we have 3 isolated sub-systems completely incapable of interacting; it is as if we have 3 separate disconnected computers.

Of course, it is desired that User-1 and User-2 should be able to communicate with the Log-manager. However, it is not desired that log entries from User-1 (and data that these reference) should be able to be transmitted to User-2 via the Log-Manager. Instead of acting like completely separate computers, we'd like it to appear there is a FIFO connecting User-1 and the Log-Manager and a separate FIFO connecting User-2 and the Log-Manager and that the data in these FIFO's are read-only.

We can do this as follows: For each user, the Log-Manager creates an additional principal; this principal can be thought of as a proxy for the Log-Manager in its interactions with each user. Thus we have 2 new principals: Log-Manager-Acting-for-User-1 and Log-Manager-Acting-for-User-2. In addition, the Log-Manager creates two new compartments (User-1-Log-Manager-Compartment, User-2-Log-Manager-Compartment) to support the interaction between the log manager and each of the users. As the owner of these compartments, the Log-Manager grants itself the right to create new gates in these compartments.

We next consider the significant operations in the interaction. These are 1) Create a new log-entry data structure and 2) Add a log-entry to the log data-structure, represented by the Create-Entry and Add-Entry generic functions. We also impose access rules that only allow Log-Manager-Acting-for-User-1 to call Create-Entry and Log-Manager-Acting-for-User-1 to allocate a log-entry in User-1-Log-Manager-Compartment (and similarly for User-2). We will refer to these compartments and principals as "satellites". Notice that we are using a defense in depth strategy: To build a new entry requires calling Create-Entry, but Create-Entry needs to call make-instance (and its subroutines); unless a principal is granted access to both generic-functions it cannot even build an entry.

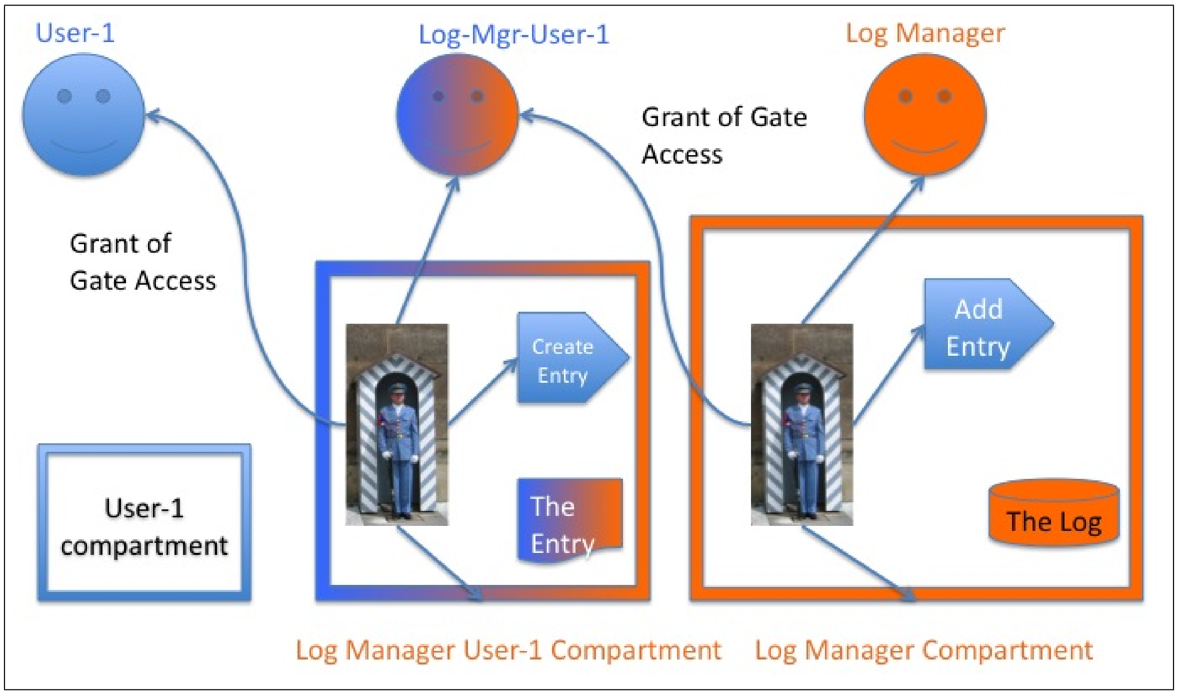


Figure 39: Pattern for limited interactions

Nevertheless, at this point the components are still isolated since Create-Entry can only be called by the Log-Manager-Acting-for-User-1 principal. In order to allow the thread acting on behalf of User-1 to actually build an entry, the Log-Manager must therefore create a Gate whose procedure is Create-Entry and whose principal and compartment are those belonging to Log-Manager-Acting-for-User-1; this gate is created in the Log-Manager-Acting-for-User-1 compartment. Finally, the Log-manager grants the User-1 principal the right to invoke gates in the Log-Manager-Acting-for-User-1 compartment. Given these structures, User-1 can invoke the gate, temporarily switching to the satellite principal and compartment; acting as the satellite principal the thread calls Create-Entry, allocating a new log entry in the satellite compartment. This is shown in Figure 39. Notice in the figure that the Log-Manager's compartment also contains a gate whose procedure is Add-Entry, and whose compartment and principal are those of the Log-Manager. The Log-Manager grants the principal Log-Manager-Acting-for-User-1 the right to invoke this gate. After Create-Entry builds the new log-entry, it invokes this gate, switching to the principal and compartment of the Log-Manager itself, and adding the new entry to the log. At this point, both the first and second gates return, restoring the principal and compartment to that of User-1.

A bit of consideration shows that the structure we have created provides only the most limited information flows among the users and the Log-Manager. The only operation that User-1 can perform on its satellite compartment is to allocate new log entries and pass them on to the log manager. Neither User-1 nor the Log-Manager have write permission to their shared satellite compartment. Thus, the Log-Manager cannot act as a vehicle for leaking information from User-1 to User-2.

We refer to the pattern illustrated above as the "hub and satellite" pattern and it is a very common way of structuring interactions among the components of a software system where highly controlled information flows are desired. This pattern illustrates how we fulfill each of the 3 principles mentioned above:

1. **Complete Mediation**: All generic functions are controlled by access rules. Every operation is monitored.
2. **Least Privilege**: Each principal is granted the most limited privileges it needs to get its jobs done. For example, the Log-Manager can only add entries to its log; it cannot access information in users' compartments. Users similarly are never granted access to Log-Manager data; they are only allowed to create entries and pass them to the Log-Manager.
3. **Separation of Privilege**: In order to add an entry to the log, one must have permission both to call Create-Entry and to allocate objects in the satellite compartment. Both checks would have to be bypassed by an attacker.

7.1.4 Implementation Techniques

As suggested in the previous section, the abstract model is implemented using the tools provided by CLOS and in particular by the MOP. The main techniques are as follows:

- All classes inherit from a common base class that provides a slot for the compartment of the object.
- Access rules are compiled (in an obvious manner) into methods whose qualifier is :permitter.
- We define a new method combination, that is essentially an OR method combination, except that it fetches :permitter methods.

  Corresponding to each original generic function, a new generic function is generated (with an uninterned mangled name) and that uses this method combination. Thus the template for each of these checking functions is:

  ```
  (or
   (eql :permitted
        <list of :permitter-method calls>)
   (error ...  ))
  ```

- We define a new method combination, that is used by all generic functions. The combined method that is built first checks the access control rules and then calls the normal elements of the combined method. The template for each combined method is:

  ```
  (apply <corresponding-gf>
         <the thread's principal>
         (mapcar #'compartment arguments))
  <rest of normal method combination>
  ```

> It would have perhaps been more elegant to use the MOP to control how the :permitter methods are fetched, in particular to fetch them based on the types of the compartments of the arguments rather than the types of the arguments. The MOP provides an entry for doing this, compute-discriminating-function as well as lower level entry points, compute-applicable-methods and compute-applicable-methods-using-classes. In some implementations there is a cache of previously computed applicable (combined) methods and this is checked in compute-discriminating-function; if the applicable method is already in the cache, then there is no need to call compute-applicable-methods. However, we only want to change how each actual argument is used to fetch applicable methods, i.e. we want to provide the ability to substitute a "argument for dispatching" for each actual argument. This must, therefore, be done within compute-discriminating-function before checking the method cache. Were this not true, we would have only modified the lower level compute-applicable-method.

The current principal and the current compartment are represented as slots in a special object that represents the "machine state". Gates are implemented as instances of a subclass of funcallable-objects. When invoked the gate, "rebinds" the compartment and principal slots of the machine state, runs its code and then restores the machine state. Rebinding the machine state, is done by saving the machine state in internal slots of the gate and modifying the appropriate slots of the object representing the machine state and then reversing these steps on exit. Of course, the compartment and principal slots of the machine state should not be modifiable by any other method. This is achieved by wrapping the accessor methods for these slots with special, implementation dependent code, that checks who is calling the accessor and signaling an error unless the caller is a gate.

7.1.5 Dynamic Access Rules

We have so far, for the most part, been assuming that the set of compartments, principals, and access rules is static. However, in Section 7.1.3 we talked about creating such entities on the fly (*e.g.* we discussed the idea of the Log-Manager creating new satellite compartments and principals and access rules governing them). There are several reasons why the situation cannot be completely static, including:

- As illustrated in section 7.1.3 many components act like servers; as new clients enter the system (*e.g.* new users log in, new web sessions are opened, etc.) there is a need to create the appropriate infrastructure to serve their needs while preserving the desired inter-client isolation.

- It is imperative that there be mechanisms for revoking access rights that have been extended to bad players. Such revocation is inherently a dynamic operation.

- We would like the system to be dynamic in the sense that it should be possible to introduce new services, applications, etc. while the system is running. These will need to dynamically instantiate a set of principals, compartments and access rules as part of their startup transient.

However, we do not want this dynamism to become a back door for subverting existing controls. In particular, if anybody is allowed to add or remove a new access rule, then this could be used to deny legitimate services or to extend illegitimate privileges to some set of users. In effect, the protection system itself could easily become the locus of attack. We, therefore, need a way to allow dynamism but to impose constraint on that dynamism; we also need to ensure that these mechanisms are not bypassable.

To achieve these goals we extend the existing framework by noting that access rules are implemented as methods and that the MOP provides generic functions that implement the process of adding and removing methods, in particular: Add-Method and Remove-Method. Since these are generic functions, they can have access rules applied to them,[12] thereby limiting who can change which types of access rules.

Generic functions are objects and therefore fall into a class system. This allows us to make all generic functions in an application, for example, inherit from a single base-class. In addition, principals are also objects falling within the class hierarchy, so we can create Principal classes for the developers (and users) of the application. This allows us to compactly specify who is allowed to change the methods implementing the generic functions making up the application, by providing :permitter methods that use these base classes as method specializers.

During the development period of the application, it is convenient to extend blanket rights to all developers to change any generic functions that are part of the application. After deployment, however, we might want to change who has this right, limiting it to a subclass of developers empowered to patch the running system. Each of these can be easily and compactly specified.

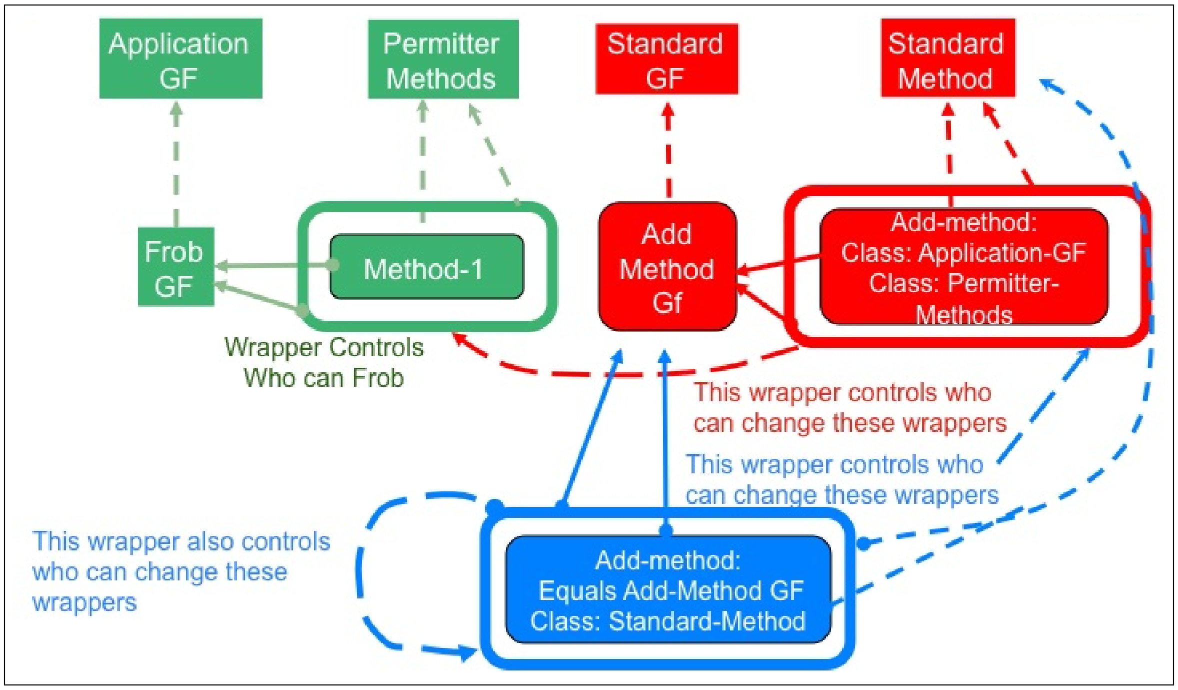

Application GF
Permitter Methods
Standard GF
Standard Method
Frob GF
Method-1
Add Method Gf
Add-method: Class: Application-GF Class: Permitter-Methods
Wrapper Controls Who can Frob
This wrapper controls who can change these wrappers
This wrapper controls who can change these wrappers
This wrapper also controls who can change these wrappers
Add-method: Equals Add-Method GF Class: Standard-Method

Figure 40: Controlling Dynamism

At this stage, we have 1) A set of access rules governing who can perform which generic functions (these are implemented as :permitter methods on the application generic functions) and 2) A set of access rules governing who can change these generic functions (these are implemented as :permitter

[12]In the current implementation this isn't strictly true, since we actually only control generic functions whose meta-class is a special class of our design. Add-method and remove-method are standard generic functions. We deal with this by using an :around method whose specializers are all T; this calls out to a generic function of the right meta-class.

methods on Add-Method and Remove-Method).

The next step in the process is to create a set of access rules governing who can change the access rules. Like all access rules, these are implemented as :permitter methods. Furthermore, like the access rules that govern who can change generic functions, these are also methods on Add-Method (and Remove-Method). To see why, recall that Add-Method takes two arguments: A generic-function and the method to be added. In the normal case, the first argument is a generic-function implementing application functionality and our access rule limits who can add (remove) new methods to that generic function. However, if the generic-function is itself Add-Method (or Remove-Method), then adding a new :permitter method controls who is allowed to add (or remove) methods from the generic-function Add-Method (or Remove-Method). But the access rules governing who can change access rules are, in fact, :permitter methods for the Add-Method (Remove-Method) generic-function. Thus imposing the correct :permitter methods controls who can change access rules. The meta-circularity closes the loop as a final set of access rules controls who can add (or remove) :permitter methods to Add-method. This is shown in figure 40. In particular, this last :permitter is applicable to itself; once in place it says that it cannot be removed and that no other such method can replace or override it.

The consequence is that once the final :permitter method is in place it establishes a chain of non-bypassable protections. It protects itself and in turn it protects the access rules that limit who can change access rules. These in turn control who can change the methods that implement application generic functions and the :permitter methods that control who can invoke which application generic functions.

In addition to the mechanisms already discussed, it is necessary to develop conventions governing which principals are allowed to impose what kinds of access rules. As we saw in Section 7.1.3 there is often a natural notion of a system component (*e.g.* the Log-Manger) owning a set of compartments, particularly compartments that it has created. Obviously, this component should have the unique right to specify access rules over data in these compartments. But can this component (*i.e.* the Principal representing the component *per se*) delegate this authority to other principals and if so, to which ones. This is still the subject of future development and requires further elaboration of our policy model.

7.1.6 CommonLisp Presents Challenges

In section 7.1.5, we showed that we can limit the degree of dynamism involved in changing access rules. But we have been ignoring rather extreme weaknesses of CommonLisp that allow the model to be subverted. These weaknesses include:

- Not all CommonLisp data are class instances operated on by generic functions. In particular, list structures are built from CONS cells which provide no slots to represent their compartment. Other immediate data (*e.g.* FIXNUMS) similarly have no easy way to represent compartments. This violates the basic assumption of our implementation that all data live in compartments and are only operated on by generic functions that can be "wrapped" with :permitter methods. It would be possible to associate a compartment with such structures using weak hashtables. This works for everything but numeric data, where we might want to distinguish the integer 1 in compartment A from the integer 1 in compartment B. Unfortunately, all 1's are both equal and eq to one another so there is no way to distinguish them without boxing them into larger structures; in the actual TIARA, this would be addressed using the hardware metadata tags.

- Normal user code can easily call internals of the language system implementation. The package system is the closest thing to a module system provided, but internal symbols of any package can be found and invoked by any code. Worse yet, the implementations normally provide reasonably good tools for discovering the internal functions in a package.
- Key internal data-structures of the CommonLisp language implementation may be built from such data and may be operated on by non-generic functions that cannot be wrapped. When combined with the previous point, it becomes reasonably straightforward to find and change the internal data structures of method combination, method caching, etc. Indeed, generic-function-methods returns a list of methods that can be modified.
- Function cells of symbols can be accessed and overwritten; this is true for generic functions as well as normal functions. This means that an attacker can inject code that could, for example, change the function-cell of Add-method, bypassing our entire scheme.
- Any function, including a generic function, can be Advised.
- The slot-access protocol includes a very low level interface, standard-instance-access, that cannot be further specialized.[13] This can be used to bypass our higher level wrappers.

These are not insurmountable problems, but dealing with all of them would require much effort. Many of the problems enumerated above would go away if every object (including immediate data) were to have a compartment and if every function were a generic function. Under such conditions, the techniques illustrated in Section 7.1.3 could be applied systematically to the entire language implementation.

7.1.7 Summary

In this section, we have introduced an access control model for Lisp-like languages. The key elements of this model are compartments, principals, access rules and gates. We presented an illustration of how this model can be used to achieve several of the principles guiding secure system construction stated by Saltzer and Schroeder (least privilege, separation of privilege, complete mediation). We presented an implementation technique in which all objects are extended to include an extra piece of metadata, the compartment and in which access rules are compiled into wrappers methods that limit access to generic functions based on the compartments of the arguments and the current principal. In the last two sections we described how the extreme openness and dynamism of CommonLisp make it difficult to implement the model in a completely non-bypassable manner and how novel tagged hardware can address this problem.

7.2 The Plan Layer and Execution Monitoring

As we have seen, the access control layer provides a set of protections based on controlling which Principals can perform operations on data in specific compartments. This is a relatively low-cost enforcement mechanism that provides relatively broad coverage. In this section, we explore another level of enforcement based on comparing the actual execution of an application to that specified in an architectural model. An architectural model can be thought of as a high level, abstract program that specifies the allowable data and control flows as well as constraints on the input and output conditions of each component within the model. The building blocks of this modeling language

[13]We thank one of the reviewers for calling this to our attention.

are hierarchically nested components, each with pre and post-conditions. These are connected by control-flow and data-flow links. Finally the modeling language includes branch and join primitives to model conditional execution. A graphical rendition of such an architectural model is shown in Figure 41. This modeling language is based on the Plan Calculus of the Programmer's Apprentice project [54].

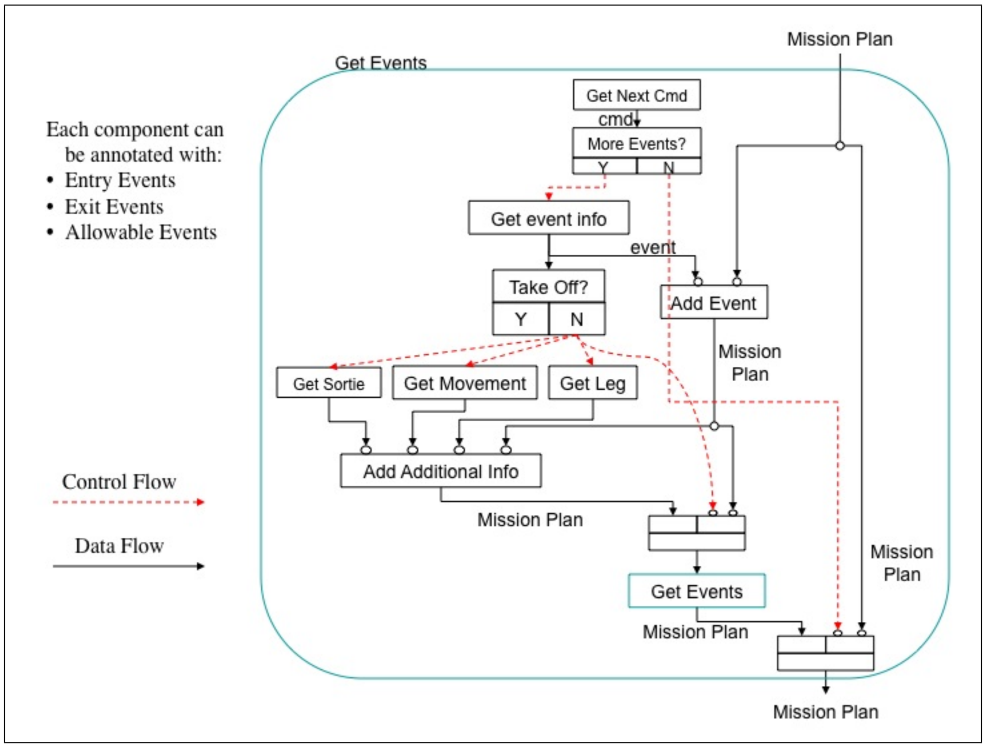


Figure 41: An Architectural Model

The core idea of the execution monitor is to compare the behavior predicted by such a model to the actual behavior produced by the program. To do this we wrap the methods of the application system with "noticer" methods that capture the inputs passed to each method as it is invoked and similarly captures the outputs produced by the method as it returns. These noticer methods then pass entry and exit events to the execution monitor together with the captured values. On entry events, the execution monitor checks whether entering the module in the event is consistent with the control flowed in the model and whether the data passed to the module is consistent with the model's data flow links. It also checks whether the data passed in is consistent with the preconditions for that module. Similarly, on output the execution monitor checks whether the actual outputs satisfy the postconditions of the model. This process of comparing actual to expected behavior is referred to as Architectural Differencing and is outlined schematically in Figure 42. The details of this process are described in detail in [68].

As noted above, the execution monitor is driven by a series of events that are produced by "noticer" wrappers that are imposed on the methods of the application. This is done automatically by a "Monitor Generator" which consults the structure of the architectural model and creates the required wrappers for those methods that correspond to modules in the model. Since the architectural model is more abstract than the program, many application methods never need to be wrapped because they occur at a level of detail lower than that described in the model. Noticer

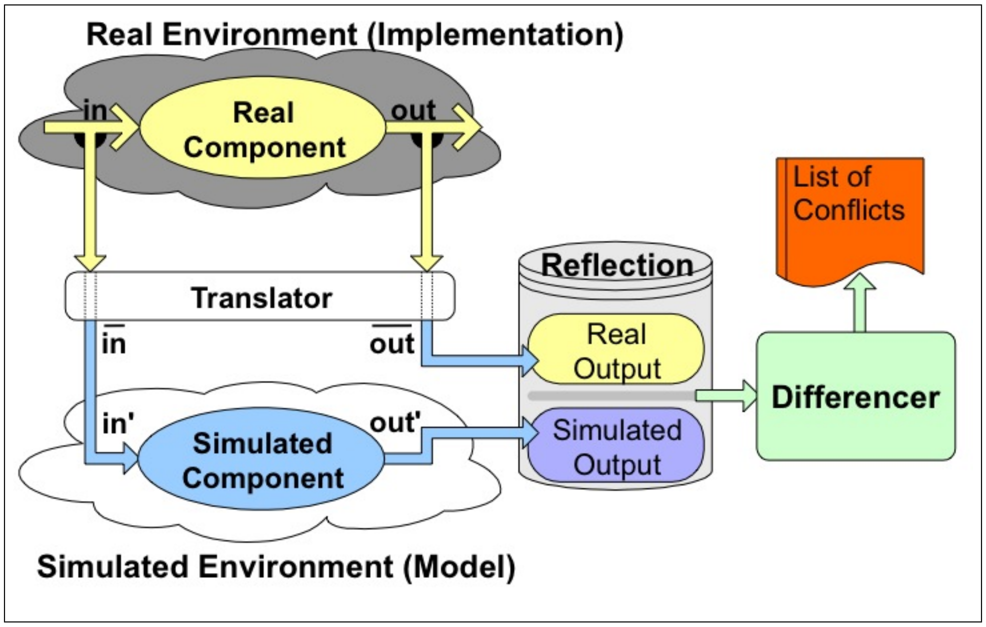


Figure 42: Architectural Differencing

wrappers are relatively simple; they capture the inputs of the method, signal an "entry" event to the execution monitor containing the input values and the name of the corresponding module in the model. They then pass control to the actual method and finally capture the output values, signal an output event to the monitor and then return control to the method's caller.

If the application's behavior is, in fact, consistent with the constraints of the architectural model, then execution proceeds normally. However, in the background the execution monitor builds a dependency structure. This dependency structure includes a set of links tracing how the preconditions of each module were satisfied as well as a second set of links connecting the outputs of each module to an assertion stating the module was believed to have behaved correctly. The execution monitor can also be programmed to produce snapshots of the data state of the application at various points in its execution.

If the execution monitor detects a violation of the constraints of the architectural model, then it initiates a diagnostic process to try to determine the cause of the violation. It does this by consulting the dependency structure it built while monitoring the execution up to this point. It also consults alternative behavioral models within the architectural model, looking for hypothesized failures that can explain the misbehavior. As explained in [68] this involves a diagnostic process that produces a Bayesian model of possible attacks, leading to possible compromises of system resources that, in turn, explain the failure.

The net result of this diagnostic analysis is a set of candidate system resources that might have been compromised together with estimates of the probability of each compromise. These are combined with prior estimates to produce a posterior "trust model" indicating which system resources are likely to be compromised and the degree to which the compromise is believed to have occurred.

This trust model can guide system and application software in making adaptive choices about how to achieve its goals. If there is more than one way to render a service, then the system should attempt to use that method which is most likely to achieve the goal as well as possible, to maximize

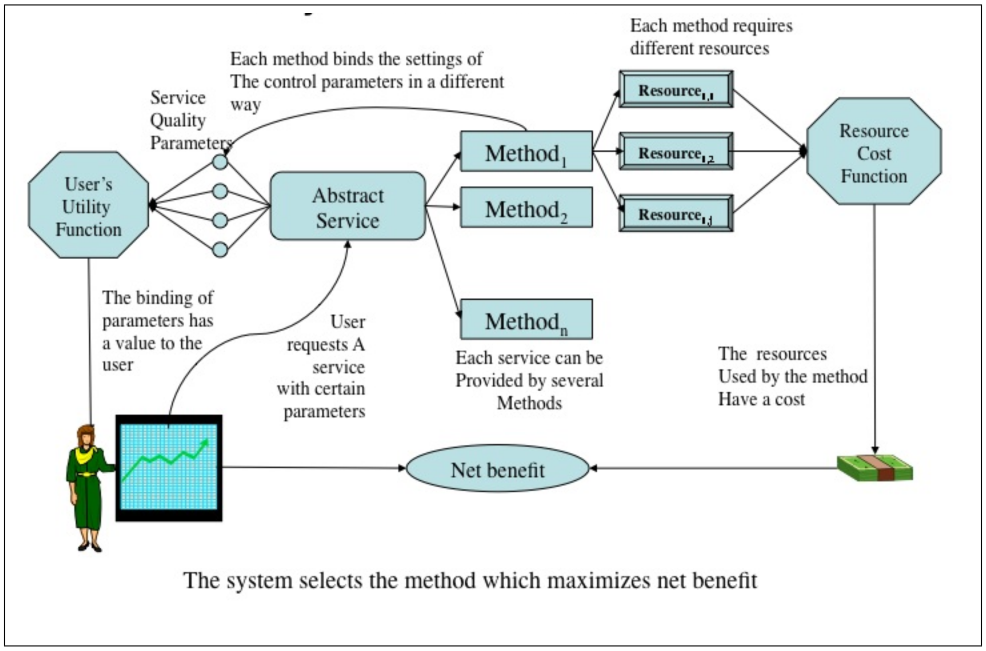

Each method requires different resources
Each method binds the settings of The control parameters in a different way
Service Quality Parameters
User's Utility Function
Abstract Service
Method$_1$
Method$_2$
Method$_n$
Resource$_{1,1}$
Resource$_{1,2}$
Resource$_{1,j}$
Resource Cost Function
The binding of parameters has a value to the user
User requests A service with certain parameters
Each service can be Provided by several Methods
The resources Used by the method Have a cost
Net benefit
The system selects the method which maximizes net benefit

Figure 43: Adaptive Software

the qualities of services that the user cares about and to minimize the risk of using resources compromised in a manner that will affect these aims. This relies on a set of decision theoretic calculations drawing on the estimates in the trust model as shown in Figure 43. More details of this can be found in [68].

This layer provides a much more extensive degree of protection than the simple access controls of the previous layer, but with more cost. It also provides for a much higher degree of "accountability" through the dependency networks constructed during execution monitoring and diagnosis and captured in the trust-model. Thus the system is capable of accounting for why it believes that a computation is proceeding either normally or abnormally. It can also account for why it believes that computational resources are compromised (and for the degree to which it believes this). Finally, the system is capable of accounting for choices it makes about how to achieve its goals when it has more than one method available. In the next section, we turn to the question of accounting for information flow through a computation.

7.3 The Data Accountability Layer and Computation Trees

The third facility provided at this level is intended to deal with accountable information flow, the central goal of the National Intelligence Community Enterprise Cyber Assurance Program (NICECAP) program and the TIARA project. In this section, we will describe a facility that builds "computation trees", dependency networks that explain how a particular value was calculated. A computation tree is a directed, labeled, acyclic graph in which the nodes are values computed during a computation, the arcs represent information flows and the labels identify computational operations. Figure 44 shows a fragment of a simple computation tree.

This can be read as saying that the value 3 was computed by adding (operation "+") two inputs: The first is the value 2 which was, in turn, computed by applying the "xx" procedure to the value "A" which was a primary input (labeled "premise" in the figure). The second input to the "+" operation was the value 1, which was produced by applying the "yy" procedure to primary input

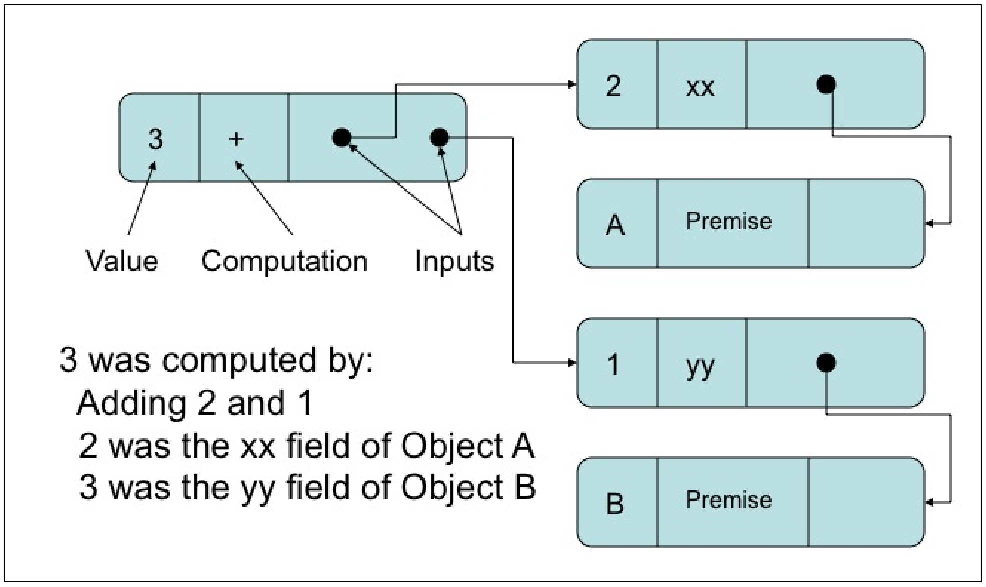


Figure 44: A Fragment of a Computation Tree

B.

Computation trees can be very large if they trace absolutely every operation performed in a computation. Thus, it is important to be able to control the granularity at which the trees are built. One obvious level is that of the method call (and indeed this is the level at which the execution monitoring facility described in Section 7.2. However, this can be overly coarse grained, so in our design we provide the flexibility to treat any method as a "black box" operation or to look within its operation. In the first case, we simply build a link for each output of the method connecting it to the inputs of the method; we label this link with the name of the method as shown in Figure 45. Such methods are called "opaque methods"; notice that if a method is treated as opaque, then every output of the method is linked back to every input even if that output actually did not depend in any way on the value of each input. Also, in treating a method as opaque, we make a conscious decision not to build computation tree structures tracing the flow through that method and through any other method that is invoked within its execution.

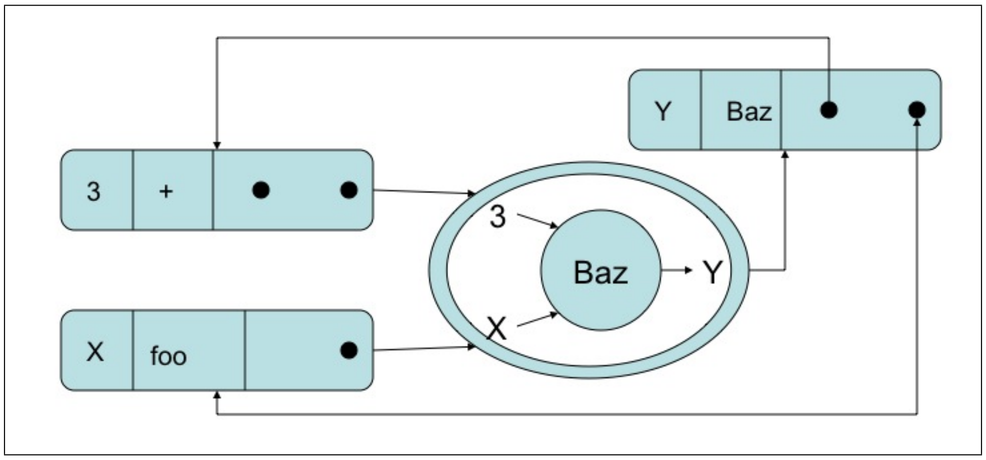


Figure 45: The Computation Tree for An Opaque Method

Treating a method as opaque is often a reasonable choice, but sometimes we want to look more microscopically and trace the flows within the method and within the method it calls. Such methods are called “transparent methods”.

Computation trees are constructed using a special data type, called a “boxed value”; these are, in fact, the nodes shown in Figure 44. “Boxed-value” is a hardware data type of the STA hardware and is treated specially by all primitive operations such as the arithmetic instructions. Unlike arithmetic data-types, boxed-values are non-immediate data (i.e. objects) with three fields:

1. The value
2. The operation-name
3. A dependency list, which is a set of other boxed-values

Recall that the STA hardware checks the data types of the operands to every instruction and signals a trap if the data types are not appropriate for that instruction. The specific trap signaled depends on the data types of the operands. Thus, for example, if an add instruction is attempted with two boxed-values as the arguments, the hardware will trap to a specific trap handler. This trap-handler then fetches the value fields of the two boxed-values, reissues the trapped instruction, collects the result, and then creates a new boxed-value whose value field is the result, whose operation-name corresponds to the trapped instruction, and whose dependency list is the two boxed values passed to the original instruction. This boxed value is then returned as the value of the trapped instruction; i.e. the boxed-value is written into the destination register of the instruction and control returns to the next instruction. This is exactly parallel to the way generic arithmetic is handled in the Lisp Machine [28] when an arithmetic instruction receives non integer operands.

In short, computations applied to “normal values” just compute, while computations applied to “boxed values” compute and trace. Opaque methods then are implemented as wrappers that intercept the inputs; if the inputs are boxed values, the wrapper unpacks the value fields of these values and then passes the raw values into the method. Since the values passed in are not boxed values, the opaque method simply computes as normal. The wrapper intercepts the outputs, creates a new boxed-value for each output as shown in Figure 45, where the output “Y” is encapsulated as a boxed value, depending on the computation “Baz” and the inputs “x” and “3”. In effect, the wrapper does for the whole method what a trap handler does for an individual operation.

There is one other issue that needs to be addressed which is how boxed values interact with method dispatching in our object system. Normally, methods are selected based on the data types of the arguments. However if a boxed value is passed to a method, we want the dispatching to be done based on the data type of the value held in the boxed value data structure. This is done by a simple modification to the method dispatching algorithm which before calculating the effective method to gain control, first unboxes the values.

7.3.1 Experimental Implementation

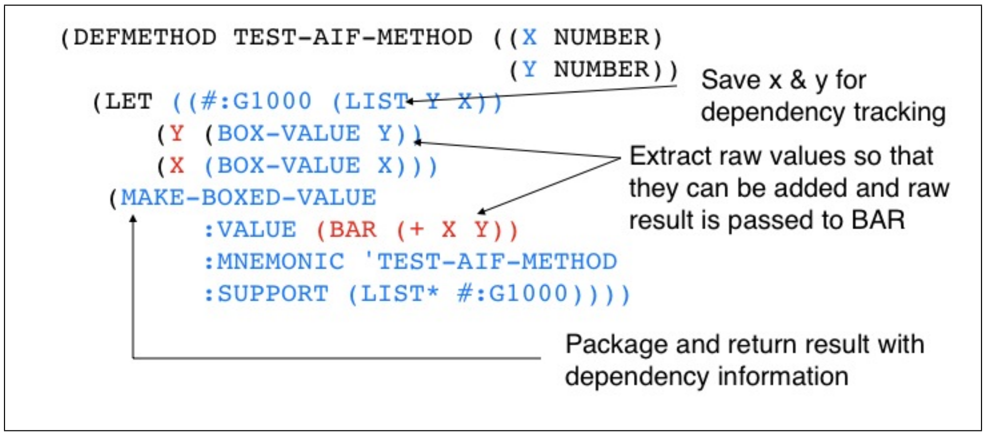
We have implemented a version of this idea following the lisp-embedding strategy style described in Section 7.1. There are two parts to the implementation strategy: The first consists of the wrappers for opaque methods; these are straightforward wrapper methods that are generated as macro expansions. Thus, for example:

```
(defmethod foo :opature (a b) (+ a b))
```

is expanded into:

```
(defmethod foo (a b)
 (let ((a (if (boxed-value? a) (value a) a))
       (b (if (boxed-value? b) (value b) b)))
   (make-boxed-value
     :value (+ a b)
     :operation 'foo
     :dependency-set (list a b))))
```

Implementing transparent methods is more difficult given that we don't yet have the STA hardware. Instead of relying on the hardware to issue traps when primitive operations are applied to boxed values, we instead perform a source-to-source code transformation. The result of this is code that simulates the hardware trapping, unpacking data around primitive operations and then repackaging the results into new boxed-values data structures. Figure 46 shows an example of the this transformation.



```
(DEFMETHOD TEST-AIF-METHOD ((X NUMBER)
                            (Y NUMBER))
  (LET ((#:G1000 (LIST Y X))
        (Y (BOX-VALUE Y))
        (X (BOX-VALUE X)))
   (MAKE-BOXED-VALUE
        :VALUE (BAR (+ X Y))
        :MNEMONIC 'TEST-AIF-METHOD
        :SUPPORT (LIST* #:G1000))))
```

Save x & y for dependency tracking
Extract raw values so that they can be added and raw result is passed to BAR
Package and return result with dependency information

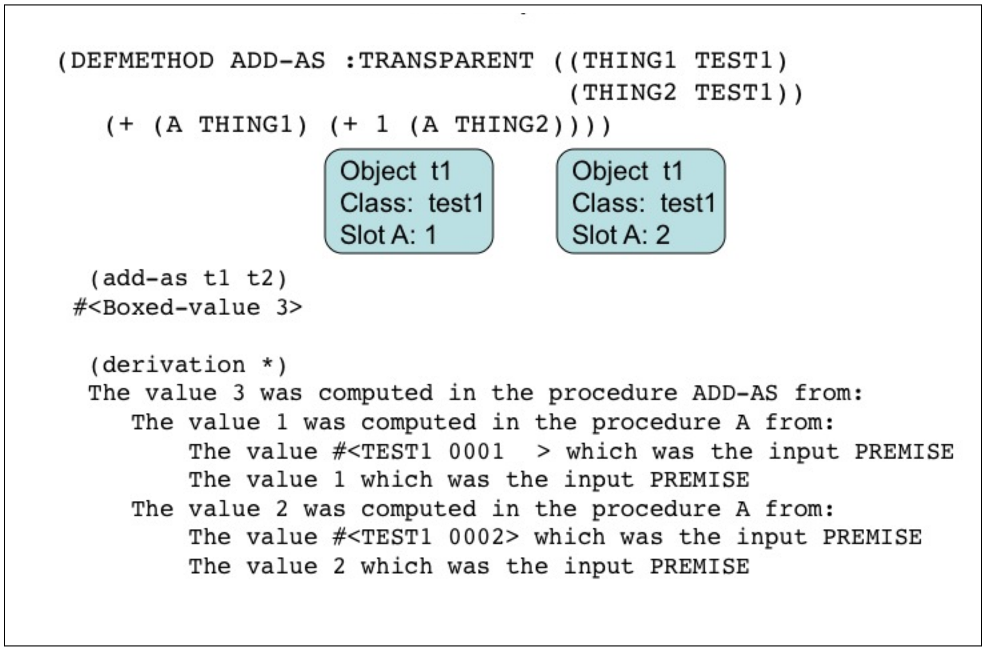
Figure 46: The Transformed Code for A Transparent Method

One advantage of building computation trees of this type is that we have a ready explanation of how a value was computed. This is easily presented in a readable manner just be tracing back through the dependency sets in either printing in a nested structure as shown in Figure 47 or by using a graphing facility to produce a graphical presentation.

These nodes of the computation tree can be logged using the logging facility described in Section 4.2.1. These log entries, however, preserve the dependency structure between the nodes, allowing replay in dependency order as well as chronological order. Figure 48 shows an example trace taken from the example described in Section 7.1.2. This trace shows the information flow through the process of computing the threat level experienced by a military unit in the application.

7.4 Summary of Upper Level Software Facilities

In this section we have describe three separate facilities used to support information flow accountability at the application middleware level. The first of these allow us to impose access controls at a level convenient for the application programmer, namely at the granularity of individual methods. The second of these allows us to impose the constraints of an architectural model on the execution



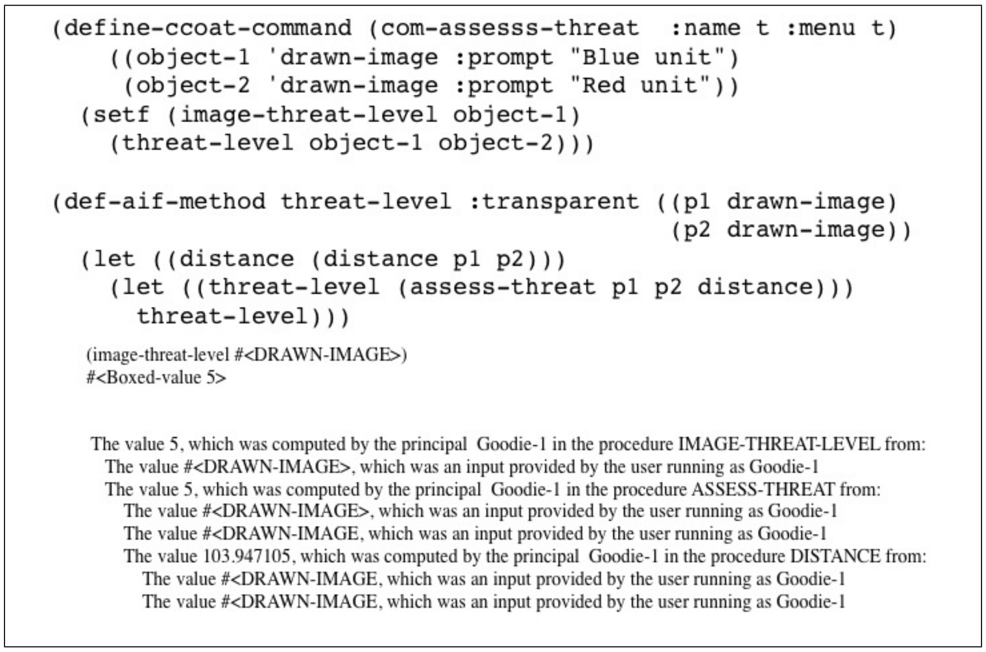
```
(DEFMETHOD ADD-AS :TRANSPARENT ((THING1 TEST1)
                                (THING2 TEST1))
   (+ (A THING1) (+ 1 (A THING2))))
```

```
Object t1        Object t1
Class:  test1    Class:  test1
Slot A: 1        Slot A: 2
```

```
  (add-as t1 t2)
 #<Boxed-value 3>

  (derivation *)
  The value 3 was computed in the procedure ADD-AS from:
     The value 1 was computed in the procedure A from:
         The value #<TEST1 0001  > which was the input PREMISE
         The value 1 which was the input PREMISE
     The value 2 was computed in the procedure A from:
         The value #<TEST1 0002> which was the input PREMISE
         The value 2 which was the input PREMISE
```

Figure 47: A Presentation of A Computational Trace



```
(define-ccoat-command (com-assesss-threat  :name t :menu t)
    ((object-1 'drawn-image :prompt "Blue unit")
     (object-2 'drawn-image :prompt "Red unit"))
  (setf (image-threat-level object-1)
    (threat-level object-1 object-2)))

(def-aif-method threat-level :transparent ((p1 drawn-image)
                                           (p2 drawn-image))
  (let ((distance (distance p1 p2)))
    (let ((threat-level (assess-threat p1 p2 distance)))
      threat-level)))
```

```
(image-threat-level #<DRAWN-IMAGE>)
#<Boxed-value 5>

The value 5, which was computed by the principal  Goodie-1 in the procedure IMAGE-THREAT-LEVEL from:
  The value #<DRAWN-IMAGE>, which was an input provided by the user running as Goodie-1
  The value 5, which was computed by the principal  Goodie-1 in the procedure ASSESS-THREAT from:
    The value #<DRAWN-IMAGE>, which was an input provided by the user running as Goodie-1
    The value #<DRAWN-IMAGE, which was an input provided by the user running as Goodie-1
    The value 103.947105, which was computed by the principal  Goodie-1 in the procedure DISTANCE from:
      The value #<DRAWN-IMAGE, which was an input provided by the user running as Goodie-1
      The value #<DRAWN-IMAGE, which was an input provided by the user running as Goodie-1
```

Figure 48: A Computational Trace From the CCOAT Application

of an application, allowing us to guarantee that the application behaves as intended. The third facility builds computation trees that trace how values were computed. Each of these facilities interface to the ZKOS log manager in order to produce persistent traces.

We have followed a strategy in each of these of using the object system of ZKOS as a key implementation vehicle. The ZKOS guarantees isolation and compartmentalization serve as a root of trust for these higher level facilities. The STA hardware is then also used to provide a root of trust both for ZKOS and to enable important features (such as the boxed-values just discussed) at the higher higher. As we have shown, these higher level facilities use the levels below to guarantee that their record keeping and execution controls cannot be bypassed even if an attacker gains access to the system.

8 Conclusions and Next Steps

In this report, we have summarized our work on the early design of the TIARA system. We have focused on three large areas of design: STA hardware, ZKOS operating system, and application infrastructure. We have shown that:

- It is possible to represent significant metadata such as data types and compartments at a very fine-grain level.
- It is possible to use modest amounts of hardware to actively enforce the semantics of this metadata without impacts on performance.
- It is possible to radically restructure the ways in which operating systems are structured so that there is no single all privileged kernel that serves as a single point of failure.
- It is possible to construct the OS so that its components are mutually suspicious and protect their key data from one another. We have shown that there are simple and nearly universal design patterns that embody this practice.
- It is possible to use hardware interlocks to guarantee that protections provided at the operating system level cannot be bypassed.
- It is possible to provide the OS with all the protections needed using hardware-supported compartments and hardware-enforced access rules controlling their use. There is no need to use a virtual memory barrier and its high context overhead to protect and isolate the OS.
- It is possible to structure and protect application-level software using the same design patterns as used for ZKOS.
- It is possible to provide an application infrastructure that enforces access control rules at the granularity of the individual method, that enforces the data flow and control flow constraints and invariants of an architectural model, and that captures the provenance of application values using computation trees.
- It is possible to use the facilities of the STA hardware and ZKOS operating system to guarantee that these application infrastructure facilities are not bypassable.

So far our investigations have involved coarse-grained implementations largely by embedding computational models of the ZKOS software within an existing (CommonLisp) environment. This implementation has been highly useful and revealing; it has demonstrated the promise of our approach and has made clear that there are no apparent "show stoppers". Rather, as we have used these tools to delve deeper into the design, we have exposed more opportunities to fundamentally change the game.

Nevertheless, there are still significant investigations to be performed and further details to be clarified. These include:

- Hardware simulator – We need to build an implementation of the ZKOS system that is accurate to the instruction set architecture of the STA hardware. We already have a simple, but adequate compiler to generate code at this level. We have also begun the implementation of a hardware simulator that will produce cycle-by-cycle accurate results. This is the most immediately pressing next step to be completed. Using this simulator we will be able to perform "micro benchmarks" on representative code samples in order to demonstrate rule-based security and quantify the impact on performance.

- ZKOS Allocation and authorization: In many of discussions of ZKOS, we have referred to an allocation service provided at a very primitive level (represented as pseudo-instructions). We have a reasonable idea of how the lowest levels of the allocator will work because of its intimate connection with the Garbage collector. However, we have not spelled out in detail the rules governing who is allowed to allocate storage in which compartments. Nor have we yet precisely defined the notions of "sub-compartments" and "sub-principals" that would allow a complete specification of the allocation policies and the authorization system that would sanction them.
- STA-level rule set format and formalization: Throughout the discussion we have referred to access rules enforced by the TMU (tag management unit) hardware. However, further work needs to be done to define the exact hardware level format of these rule sets. Furthermore, the rules enforced by the TMU hardware are very low-level cached versions of more general rules. Our actual rule specification language will, in contrast, consist of a set of procedures that when presented with specific instructions, principals, and compartments tell whether the operation is sanctioned and if so, what the metadata of the result should be. These higher level procedures are run when the TMU hardware misses, compute the answer and then cache it in a table in memory. Further work needs to be done in defining the language of the high level procedures and the in-memory cache. This language should be carefully restricted in power to ease formal analysis.
- Proofs: We have argued that TIARA is a codesign effort involving hardware design, software architecture, and verification. As we do this, we can add or modify features of the hardware and software architectures so that proofs and verification become tractable. We have further argued that much of the formal verification can be done by constructing proofs based on reachability relationships within a graph mirroring the structure of compartments, principals and access rules. Formalizing this proof methodology is an important next step in the design process.
- Demo Applications: In the current project, we have motivated our work using a simple but representative application (CCOAT) and several illustrative program fragments from other sources. Further progress requires a larger set of applications raising a more diverse set of issues.

Based on our experience and developments within the TIARA project, we recommend:

- It is time to stop being complacent about the current hardware security base. Fully developing a new architecture for deployment is a large effort that will take time and resources. It is time to get started.

- It is time to give up on the hypothesis that we can write large, error-free software systems. Instead, we should develop and employ engineered systems that can achieve the necessary trustworthiness despite the inevitable residual flaws in systems. The new hardware base suggested above makes this possible. This means moving away from relying on monolithic domain software systems as our security base. We must develop a new base system around ideas like those in ZKOS (Section 4) and build the infrastructure (tools, libraries, languages) to allow applications to be structured in this way as well.

- It is time to stop relying on systems that exceed both human and machine understanding. No single human can manage millions of lines of code, and no formal system can validate unstructured code of this size and complexity. Formal verification is now powerful enough to be used, but we must design and decompose our systems to meet the capabilities of verification.
- The hardware, OS, and verification strategy must be designed together. Engineered codesign is within the realm of state-of-the-art engineering, while analysis and assurance of unconstrained hardware and software artifacts is not.

In particular, we recommend the following specific steps:

- Complete the design of one or more STA hardware implementations. This is largely virgin territory and we should encourage diversity of approaches. For example, it might be sensible to pursue several alternative instruction set architectures (ISA) and it is certainly reasonable for any such ISA to pursue alternative micro-architectural implementations.
- Build one or more ZKOS-like operating systems. Again, this is largely uncharted waters. It makes sense to explore the various ways in which STA-like hardware can be used to support operating systems without all privileged kernels. We have illustrated an approach within our ZKOS work, but we think there are many possible design patterns that should be pursued.
- Develop a verification strategy. We have argued that one virtue of the TIARA approach is that it has considered all levels of a computer system design: hardware, system software and application middleware. We have also argued that the approach used in these investigations should be more amenable to formal verification than current systems; in particular, we believe that the verification effort will not need to look at millions of lines of code but rather hundreds of lines of access rules. It is time to develop formal methods based on this insight.
- Treat verification as a Codesign effort. Formal verification of full operating systems (not to mention application middleware systems like web browsers and database servers) are still beyond the state-of-the-art. However, we believe that this task can be made much easier by providing hardware features that provide non-bypassable guarantees upon which the formal analysis can rely. When proofs become difficult, we should consider whether there are addition such hardware guarantees that can make the proof effort much simpler. Similarly, new software system designs can erect clear boundaries on information flows that also aid the proof effort, particularly, when the software architecture has hardware guarantees as in our ZKOS design. As with the hardware, software architectures that are difficult to verify can be modified to clarify the intended separation and flow properties and to impose enforcement points that act as important structuring elements in an overall proof. This Codesign strategy requires collaboration between formal method designers and hardware architects and software system designers.
- Programming language and development environments: However our new systems are structured, we will need programming languages designed for the new approaches we have outlined. Most of programming languages have been compromises reflecting the compromises made in

hardware and operating systems of the past. We should look to new designs that are both flexible and dynamic but analyzable. Similarly, we need new programming environments that support the much broader challenges that we have outlined in this discussion.

Clearly these efforts will require a change of mind-set from one rooted in past eras where resource limitations constrained what we might do to one in which we realize that resources are abundant and should be applied to making systems worthy of the trust we place in them. The TIARA project has been a first step in this process.

9 References

[1] Jim Alves-Foss, Carol Taylor, and Paul Oman. A multi-layered approach to security in high assurance systems. In *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, page 90302.2, Washington, DC, USA, 2004. IEEE Computer Society.

[2] G. M. Amdahl, G. A. Blaauw, and Jr. Frederick P. Brooks. Architecture of the ibm system/360. *IBM Journal of Research and Development*, pages 87–101, April 1964.

[3] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *Proc. of SAS'04*, pages 100–115, 2004.

[4] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Architectural support for run-time validation of program data properties. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):546–559, May 2007.

[5] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement unix prototype. In *SSYM'95: Proceedings of the 5th conference on USENIX UNIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 1995. USENIX Association.

[6] Clark Baker, David Chan, Jim Cherry, Alan Corry, Greg Efland, Bruce Edwards, Mark Matson, Henry Minsky, Eric Nestler, Kalman Reti, David Sarrazin, Charles Sommer, David Tan, and Neil Weste. The symbolics ivory processor: a 40 bit tagged architecture LISP microprocessor. In *Proceedings of the International Conference on Computer Design*, pages 512–515, 1987.

[7] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM SOSP*, pages 164–177, 2003.

[9] John Barkley. Implementing role-based access control using object technology. In *First ACM Workshop on Role-Based Access Control*, November 30 – December 1 1995.

[10] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretations. Technical Report Tech. Report MTR-2997, MITRE Corp., July 1975.

[11] David Elliot Bell. Looking back at the bell-la padula model. *Proceedings of the Annual Computer Security Applications Conference*, pages 337–351, 2005.

[12] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, , and D. Moon. Common lisp object system specification. Technical Report 88-002R, X3J13, June 1988.

[13] Jeremy Brown, J.P. Grossman, Andrew Huang, and Jr. Thomas F. Knight. A capability representation with embedded address and nearly-exact object bounds. Technical Report Aries Project Technical Report 5, MIT AI Lab, April 2000.

[14] Jeremy Brown and Jr. Thomas F. Knight. A minimal trusted computing base for dynamically secure information flow. Technical Report Aries Project Technical Report 15, MIT AI Lab, November 2001.

[15] Internet Crime Complaint Center. 2007 internet crime report. Technical report, The National White Collar Crime Center, Bureau of Justice Assistance, Federal Bureau of Investigation, 2007.

[16] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, 1987.

[17] Jedidiah R. Crandall, Frederic T. Chong, and S. Felix Wu. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization*, 5:359–389, December 2006.

[18] R. J. Creasy. "the origin of the vm/370 time-sharing system". *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[19] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the International Symposium on Computer Architecture*, pages 482–493, 2007.

[20] Andre DeHon, Jr. Thomas F. Knight, John C. Mallery, and Howard E. Shrobe. Towards a security tagged architecture (sta). white paper prepared for NSF Safe Computing Workshop, November 2006.

[21] Rance J. DeLong, Thuy D. Nguyen, Cynthia E. Irvine, and Timothy E. Levin. Toward a medium-robustness separation kernel protection profile. In *Proceedings of the Annual Computer Security Applications Conference*, pages 40–49, 2007.

[22] D. E. Denning. A lattice model of secure information flow. *Communications of the Association of Computing Machinery*, 19(5):236–243, May 1976.

[23] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. HardBound: Architectural support for spatial safety of the C programming language. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 103–114, 2008.

[24] C. Baker et al. The symbolics ivory processor: a 40 bit tagged architecture lisp microprocessor. In *Proceedings of the 1987 IEEE International Conference on Computer Design*, pages 512–515, October 1987.

[25] J. Feiertag and Peter G. Neumann. The foundations of a provably secure operating system (psos. In *In Proceedings of the National Computer Conference*, pages 329–334. AFIPS Press, 1979.

[26] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *Proceedings of the 15th NIST-NSA National Computer Security Conference*, October 13–16 1992.

[27] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, 1990.

[28] R.D. Greenblatt, T.F. Knight Jr., J. Holloway, D.A. Moon, and D.L. Weinreb. The lisp machine. In *Interactive Programming Enviornments*. McGraw-Hill, 1984.

[29] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, 2005.

[30] Cornelius J. Haley, Sheryl M. Luera, Mary D. Schanken, and William B. Geer. Final evaluation report unisys a series mcp/as release 3.7. Technical Report CSC-EPL-871003, Library No. S-228,515, National Computer Security Center, Fort Meade, MD, August 5 1987.

[31] Intel Corporation, Santa Clara, CA. *Introduction to the iAPX 432 Architecture Manual*, 1981.

[32] A. H. Karp, R. Gupta, G. J. Rozas, and A. Banerji. Using split capabilities for access control. *IEEE Software*, 20(1):42–49, January 2003.

[33] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Number ISBN 0-201-17589-4. Addison-Wesley, 1989.

[34] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[35] Butler W. Lampson. Protection. In *Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. (reprinted in Operating Systems Review, 8,1, January 1974, pp. 18 - 24).

[36] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[37] H. Levy. *Capability-Based Computer Systems.* Digital Press, 1984.

[38] Henry M. Levy. *Capability-based Computer Systems.* Digital Press,, Bedford, MA, 1984.

[39] Howard F. Lipson. Tracking and tracing cyber-attacks: Technical challenges and global policy issues. Technical Report CMU/SEI-2002-SR-009, CMU CERT, 2002.

[40] John C. Mallery and Howard E. Shrobe. Zero-kernel secure operating system, November 2006.

[41] John Markoff. Worm infects millions of computers worldwide, January 2009.

[42] M. Miller, K. Yee, and J. Shapiro. Capability myths demolished, 2003.

[43] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of ACM Conference on Lisp and Functional Programming)*, pages 235–246. ACM, 1984.

[44] David A. Moon. Garbage collection in a large lisp system. In *Proceeding of the ACM Conference on Lisp and Functional Programming*, pages 235–246, 1984.

[45] David A. Moon. Architecture of the symbolics 3600. In *Proceedings of the International Symposium on Computer Architecture*, pages 76–83, 1985.

[46] David A. Moon. Architecture of the symbolics 3600. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 76–83, 1985.

[47] A. S. Tanenbaumand S. J. Mullender and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 558–563, 1986.

[48] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[49] Peter G. Neumann and Richard J. Feiertag. Psos revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 208–216, December 2003.

[50] Thuy D. Nguyen, Timothy E. Levin, and Cynthia E. Irvine. High robustness requirements in a common criteria protection profile. In *Proceedings of the IEEE International Workshop on Information Assurance*, 2006.

[51] Elliot I. Organick. *The MULTICS system: An examination of its structure.* The MIT Press, 1972.

[52] Elliot I. Organick. *Computer System Organization: The B5700/B6700 Series.* Academic Press, 1973.

[53] Elliot I. Organick. *A Programmer's View of the Intel 432 System.* McGraw-Hill, 1983.

[54] Charles Rich and Howard E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report Technical Report 354, MIT Artificial Intelligence Laboratory, December 1976.

[55] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, 38:39–47, 2005.

[56] John Rushby. Design and verification of secure systems. In *8th ACM Symposium on Operating System Principles*, pages 14–16, December 1981.

[57] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[58] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of*

CSFW, pages 255–269, 2005.

[59] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[60] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[61] SANS, 2009.

[62] O. Sami Saydjari. Lock: An historical perspective. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 96, Washington, DC, USA, 2002. IEEE Computer Society.

[63] O.S. Saydjari, J.M. Beckman, and J.R. Leaman. Lock trek: Navigating uncharted space. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[64] M. D. Schroeder, D. D. Clark, J. H. Saltzer, and D. H. Wells. Final report of the multics kernel design project. LCS/TR 196, MIT, June 1977.

[65] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3), March 1972.

[66] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[67] Howard Shrobe, Robert Laddaga, Robert Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, and Tim Hollebeek. AWDRAT: A Cognitive Middleware System for Information Survivability. In *Innovative Applications of Artificial Intelligence*. AAAI Press, July 2006.

[68] Howard Shrobe, Robert Laddaga, Robert Balzer, Neil Goldman, Dave Wile, Marcelo Tallis, and Tim Hollebeek. Awdrat: Architectural Differencing, Wrappers, Diagnosis, Recovery, Adaptivity and Trust management. In *Proceeding of the AAAI Conference on Innovative Applications of Artificial Intelligence*. AAAI Press, July 2006.

[69] Richard E. Smith. Cost profile of a highly assured, secure operating system. *ACM Trans. Inf. Syst. Secur.*, 4(1):72–101, 2001.

[70] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[71] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[72] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *IEEE Computer*, pages 44–51, May 2006.

[73] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture*, 2004.

[74] Steve VanDeBogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazieres. Labels and event processes in the asbestos operating system. In *ACM Transactions on Computer Systems*, volume 25, pages 1–43, December 2007.

[75] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proc. of the 14th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2008.

[76] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP Computer and Its Operating*

System. North Holland, 1979.

[77] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–315, October 2002.

[78] Nickolai Zeldovich. *Securing Untrustworthy Software Using Information Flow Control.* PhD thesis, Stanford University, October 2007.

10 List of Acronyms

ACL	Access Control List
ALU	Arithmetic Logic Unit
CCOAT	Commander's Course of Action Tool
CISC	Complex Instruction Set Computer
CLIM	Common Lisp Interface Manager
CLOS	Common Lisp Object System
CLR	Common Language Runtime
DARPA	Defense Advanced Research Projects Agency
DMA	Direct Memory Addressing
DNS	Domain Name System
DRAM	Dynamic Random Access Memory
D-cache	Data Cache
FIFO	First In First Out
GC	Garbage Collector
HCI	Human Computer Interaction
HEX	Hash Execution
I/O	Input/Output
I-cache	Instruction cache
ISA	Instruction Set Architecture
JVM	Java Virtual Machine
MAC	Mandatory Access Control
MILS	Multiple Independed Layers/Levels of Security
MLS	Multi-Level Security
MOP	Meta Object Protocol
NICECAP	National Intelligence Community Enterprise Cyber Assurance Program
OS	Operating System
PC	Program Counter
PLA	Programmable Logic Array
RISC	Reduced Instruction Set Computer
RO	Read Only
R/W	Read/Write
SELinux	Security Enhanced Linux
SRAM	Static Random Access Memory
STA	Security Tagged Architecture
TCP/IP	Transmission Control Protocol/Internet Protocol
TIARA	Trust-Management, Intrusion-tolerance, Accountability, and Reconstitution Architecture
TLB	Translation Lookaside Buffer
TMS	Trust Maintenance System
TMU	Tagged Memory Unit
USB	Universal Serial Bus
VLSI	Very Large Scale Integration
VM	Virtual Machine

VoIP	Voice over Internet Protocol
ZKOS	Zero Kernel Operating System